Incremental approach for application GUI migration using metamodels

Benoît Verhaeghe

Université de Lille, Faculté des sciences et technologies

Département de formation doctorale en informatique **École doctorale MADIS Lille**
UFR IEEA

Incremental Approach for Application GUI Migration using Metamodels

Approche Incrémentale pour la Migration des Interfaces Graphiques d'Applications utilisant les Métamodèles

THÈSE

présentée et soutenue publiquement le 21 octobre 2021

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Benoît Verhaeghe

Composition du jury

Président :	Franck Barbier (Professor – Université de Pau et des Pays de l'Adour)
Rapporteurs :	Salah Sadou (Professor – Université Bretagne Sud) Jean-Rémy Falleri (Associate Professor – Université de Bordeaux)
Directeurs de thèse :	Nicolas Anquetil (Associate Professor – Université Lille) Anne Etien (Professor – Université Lille)
Co-Encadrant de thèse :	Abderrahmane Seriai (Associate Professor – Berger-Levrault)

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
INRIA Lille - Nord Europe

Acknowledgments

First of all, I would like to thank my supervisors Anne Etien and Nicolas Anquetil, for giving me the opportunity to be part of this adventure. I also want to thank Abderrahmane Seriai and Laurent Deruelle for helping me reduce the gap between fundamental research and the industrial world. Thank you for all your support, encouragement, and advice throughout the years.

I would like to thank Franck Barbier, Salah Sadou, and Jean-Rémy Falleri for accepting to be part of my Ph.D. committee. I am honored by your presence, and I really appreciate your comments and advice.

My thesis has been done in an especially pleasant context, thanks to the RMOD team members. In particular, thanks to Julien, Cyril, Christopher, and Guillaume for the many tea and coffee breaks. I also had the chance to work in the research team of Berger-Levrault. I want to thank Pascal, Michel, Jimmy, and Julien for the time spent with me discussing movies and politics. I also had the chance to be accompanied by former Ph.D. students: JB, Clement, Sophie, Vincent B., and Vincent A., who help me express frustrations and problems. *Tot anoste keer!*

I also had the chance to live at Bruno's home during my Ph.D. manuscript redaction. It was a particularly pleasant experience, and I would never thank you enough for letting me live with you all.

I am deeply grateful to my family, who have supported me during this experience. Especially thanks to my parents and in-laws for everything they provide to me, and my brothers, Adrien and Tristan, who unconditionally supported me.

I also want to thank my beloved Annaëlle for being part of my life and sharing many unforgettable moments with me, supporting and encouraging me through these years.

Above all, I want to thank my grandmother who allowed me to live with her for 8 years. *Merci mémé*.

À Brigitte Lhomme,

Abstract

Developers use GUI frameworks to design the graphical user interface of their applications. It allows them to reuse existing graphical components and build applications in a fast way. However, with the generalization of mobile devices and Web applications, GUI frameworks evolve at a fast pace: JavaFX replaced Java Swing, Angular 8 replaced Angular 1.4 which had replaced GWT (Google Web Toolkit). Moreover, former GUI frameworks are not supported anymore. This situation forces organizations to migrate their applications to modern frameworks regularly to avoid becoming obsolete.

To ease the migration of applications, previous research designed automatic approaches dedicated to migration projects. Whereas they provide good results, they are hard to adapt to other contexts than their original one. For instance, at Berger-Levrault, our industrial partner, applications are written in generic programming languages (Java/GWT), proprietary "4th generation" languages (VisualBasic 6, PowerBuilder), or markup languages (Silverlight). Thus, there is a need for a language-agnostic migration approach allowing one to migrate various GUI frameworks to the latest technologies. Moreover, when performing automatic migration with these approaches, part of the migrated application still needs to be manually fixed. This problem is even more important for large applications where this last step can last months. Thus, companies need to migrate their application incrementally to ensure end-user continuous delivery throughout the process.

In this thesis, we propose a new incremental migration approach. It aims at allowing the migration of large applications while ensuring end-user delivery. It consists of migrating pages using our automatic GUI migration tool, fixing them, and integrating them in a hybrid application. To create our GUI migration tool, we designed a pivot meta-model composed of several packages representing the visual and the behavioral aspects of any GUI. We detailed multiple implementations of our GUI migration tool that extract and generate GUI using different frameworks.

We successfully applied our migration approach to a real industrial application at Berger-Levrault. The migrated application is now in production. We also validated our automatic GUI migration tool on several migration projects, including applications developed with programming and markup languages. The company is currently using our approach for other migration projects.

Keywords: Graphical User Interface, Model-Driven Engineering, Migration, Industrial

Résumé

Les développeurs utilisent des frameworks d'interface graphique (GUI frameworks) pour concevoir l'interface utilisateur graphique de leurs applications. Cela leur permet de réutiliser des composants graphiques existants et de construire des applications rapidement. Cependant, avec la généralisation des appareils mobiles et des applications Web, les GUI frameworks évoluent à un rythme rapide : JavaFX a remplacé Java Swing, Angular 8 a remplacé Angular 1.4 qui avait remplacé GWT (Google Web Toolkit). De plus, les anciens GUI frameworks ne sont plus supportés. Cette situation oblige les organisations à migrer régulièrement leurs applications vers des frameworks modernes pour éviter qu'elles deviennent obsolètes.

Pour faciliter la migration des applications, des recherches antérieures ont conçu des approches automatiques dédiées à des projets de migration. Bien qu'elles fournissent de bons résultats, elles sont difficiles à adapter à d'autres contextes que celui d'origine. Par exemple, chez Berger-Levrault, notre partenaire industriel, les applications sont écrites dans des langages de programmation génériques (Java/GWT), des langages propriétaires de "4ème génération" (VisualBasic 6, PowerBuilder), ou des langages de balisage (Silverlight). Il est donc nécessaire d'adopter une approche de migration indépendante du langage, qui permette de faire migrer diverses interfaces graphiques vers les technologies les plus récentes. En outre, lors d'une migration automatique avec ces approches, une partie de l'application migrée doit encore être corrigée manuellement. Ce problème est encore plus important pour les grandes applications où cette dernière étape peut durer des mois. Les entreprises doivent donc migrer leur application de manière incrémentale afin de garantir une livraison continue à l'utilisateur final tout au long du processus.

Dans cette thèse, nous proposons une nouvelle approche de migration incrémentale. Elle vise à permettre la migration de grandes applications tout en garantissant la livraison à l'utilisateur final. Elle consiste à migrer des pages à l'aide de notre outil de migration automatique de GUI, à les corriger et à les intégrer dans une application hybride. Pour créer notre outil de migration de GUI, nous avons conçu un méta-modèle pivot de GUI composé de plusieurs paquetages représentant les aspects visuels et comportementaux de toute GUI. Nous avons détaillé plusieurs implémentations de notre outil de migration de GUI qui extraient et génèrent des GUI utilisant différents frameworks.

Nous avons appliqué avec succès notre approche de migration sur une application industrielle de Berger-Levrault. L'application migrée est maintenant en production. Nous avons également validé notre outil de migration automatique d'interface graphique sur plusieurs projets de migration incluant des applications développées avec des langages de programmation et de balisage. L'entreprise utilise actuellement notre approche pour d'autres projets de migration.

Mots-clés: Interface Graphique, Ingénierie Dirigée par les Modèles, Migration, Industriel

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Problem . . . 3
- 1.3 Our Approach in a Nutshell . . . 4
- 1.4 Contributions . . . 5
- 1.5 Structure of the Thesis . . . 5
- 1.6 List of Publications . . . 6

2 State of the Art **9**
- 2.1 GUI representation . . . 9
- 2.2 GUI migration . . . 13
- 2.3 Incremental migration . . . 18

I GUI migration **23**

3 Approach **25**
- 3.1 GUI Terminology and Concept . . . 25
- 3.2 Approach overview . . . 31
- 3.3 Visual code migration approach . . . 35
- 3.4 Behavioral code migration approach . . . 46
- 3.5 Conclusion . . . 53

4 Implementation **55**
- 4.1 Visual code extraction . . . 55
- 4.2 Visual code generation . . . 61
- 4.3 Behavioral code extraction . . . 63
- 4.4 Behavioral code generation . . . 68
- 4.5 Conclusion . . . 73

5 Migration Validation **75**
- 5.1 Visual code migration validation . . . 75
- 5.2 Behavioral code migration validation . . . 83

5.3 Discussion . . . 87
5.4 Threats to Validity . . . 90
5.5 Conclusion . . . 92
5.6 GUI migration conclusion . . . 93

II Incremental migration approach **95**

6 Incremental migration **97**
6.1 Incremental Migration Approach . . . 97
6.2 Hybrid architecture . . . 99
6.3 Implementation . . . 102
6.4 Conclusion . . . 107

7 Incremental migration validation **109**
7.1 Case Study: Omaje Application . . . 109
7.2 Research Questions and Evaluation Methods . . . 111
7.3 Evaluation Results . . . 113
7.4 Incremental approach discussion . . . 117
7.5 Threats to Validity . . . 119
7.6 Incremental approach conclusion . . . 121

8 Conclusion **123**
8.1 Summary . . . 123
8.2 Contributions . . . 124
8.3 Future Work . . . 125

Bibliography **129**

List of Figures

1.1 Summary of the approach . . . 4

2.1 KDM - UIResources Class Diagram . . . 10
2.2 IFML - View Elements . . . 11

3.1 GUI example . . . 27
3.2 A layout example . . . 28

3.3 GUI migration concrete example from Java to Angular 32
3.4 Pivot meta-model 33
3.5 Our GUI migration process 34
3.6 Core package 36
3.7 Excerpt of the widget package 37
3.8 Layout package 38
3.9 Visual code extraction steps 40
3.10 Visual code generation sub-steps 44
3.11 Behavioral package 47
3.12 Behavioral code extraction sub-steps 48
3.13 Behavioral code generation sub-steps 51
3.14 Our GUI detailed migration process 54

4.1 Example of model transformation for Listing 4.6 67
4.2 Angular Event Binding feature 70
4.3 Data Binding in Angular 72

5.1 BLCore - GUI meta-model 76
5.2 Visual comparison of a page migration (Kitchensink) 81
5.3 Visual comparison of the User Setting page (Traccar) 82
5.4 Visual comparison (DBManager) 82

6.1 The incremental migration process 98
6.2 Hybrid architecture 100
6.3 Operational architecture of hybrid application for GWT and Angular 102
6.4 Packaging and using Angular migrated page with Web Component 103
6.5 Data transformation process in hybrid communication 107

7.1 Time spent by module to migrate the Visual code 110
7.2 Time spent by module to migrate the Behavioral code 111
7.3 Performance evaluation result 116

8.1 Layout manager conversion 126

List of Tables

2.1 Existing migration project 16
2.2 Hybrid architectures fulfilling constraints 20

3.1 Sub-steps to support the extraction of a new GUI framework . . . 41
3.2 Sub-steps to extract Visual code using a known framework 42
3.3 Sub-steps to support the generation using a new GUI framework . 44
3.4 Sub-steps to generate the Visual code using a known framework . 45
3.5 Sub-steps to configure new framework extraction for Behavioral code 49
3.6 Sub-steps to extract application Behavioral code 50
3.7 Sub-steps toi configure new framework generation for Behavioral code 52
3.8 Sub-steps to generate application Behavioral code 52

5.1 Case study Description 76
5.2 Application descriptions 77
5.3 Extraction results 80
5.4 Result of manual event handler extraction check 85
5.5 Natural code 86

7.1 Communications performance in millisecond 114
7.2 Building performance in second 115

List of Listings

3.1 Behavioral code in Java 29

4.1 Snippet of an GXT login view in XML 56
4.2 User interface creation in Java GWT 59
4.3 Building layout and DOM in Pharo Spec 60
4.4 Complex layout creation in Java GWT 60
4.5 Creating event handlers in Java/GWT 65
4.6 Example of Manipulation code 66
4.7 Example of Java code 70
4.8 Example of HTML code migrated from Java code in Listing 4.7 . 70
4.9 Example of TypeScript code migrated from Java code in Listing 4.7 71
4.10 Data Binding - HTML part 72
4.11 Data Binding - TypeScript part 72

6.1 Exposed (displayPhase, addDataRefresh) and not exposed (addDataRefreshEvent) methods to hybrid architecture 106

CHAPTER 1

Introduction

Contents

1.1 Context 1
1.2 Problem 3
1.3 Our Approach in a Nutshell 4
1.4 Contributions 5
1.5 Structure of the Thesis 5
1.6 List of Publications 6

1.1 Context

This thesis takes place in an industrial partnership with Berger-Levrault[1]. Berger-Levrault provides software solutions to public and private stakeholders. For instance, it builds management software, computerized maintenance management systems (CMMS), medical record and care software, teaching software solutions, and so on.

A distinctive point is the company's growth strategy. Besides of developing new software systems for their customers, Berger-Levrault acquires other companies with existing systems. Whereas this strategy has its advantages, it also impacts software development. Indeed, the acquired companies do not always use the programming language selected at Berger-Levrault. This results in a plurality of software systems developed using several programming languages and frameworks.

Fortunately, for the last ten years, Berger-Levrault has been launching a dynamic of innovation in its products, notably by integrating breakthrough functionalities resulting from research. In that sense, a research and development department was created to reduce the gap between the scientific world and the industrial world.

Among the several scientific projects conducted at Berger-Levrault, one consists of a large-scale renovation project to standardize development. It includes the

[1] Berger-Levrault: https://www.berger-levrault.com/fr/

pooling of the company's existing software solutions into one extensive and extensible system using a component-oriented architecture [Allier et al., 2011]. To achieve this goal, several projects were carried out, are currently in progress, and are planned for the future. Micro-services [Selmadji et al., 2020], micro front-end, software interoperability [Amokrane et al., 2020], software testing, and software migration are among the hot topics in software renovation at Berger-Levrault. This thesis is part of the renovation project and deals with the migration of Graphical User Interface (GUI) of its applications.

Companies, as Berger-Levrault, develop Graphical User Interface for their customers. GUI facilitates end-users interaction with a software system. To build easy-to-use user interfaces, developers use GUI frameworks. A framework consists of a group of generic functionality that helps developers creating software systems. Using GUI frameworks, developers have access to existing graphic components, such as buttons and text inputs. It allows them to develop faster and reduce the cost of creating GUIs.

However, these GUI frameworks evolve at a fast pace. More than 90 GUI frameworks were developed during the past 40 years[2]. On the one hand, the old GUI frameworks are not supported anymore: for example, the last major version of GWT was in 2009. On the other hand, modern GUI frameworks evolve fast: in 2020 three major versions of Angular[3], one major version and two minors versions of React.js[4], one minor version of Vue.js[5], and two major versions of Ember.js[6] were released. Using the most modern GUI frameworks is essential for companies. Indeed, from the end-users point of view, the GUI *is* the software system. Thus, software systems using the last GUI standards are more attractive and are a plus for companies selling them. Using modern GUI frameworks improves the code reusability, the development agility, and reduces the risk of bugs and regressions. This situation forces companies to regularly update their software systems to keep their customers and avoid being stuck in old technologies.

Approaches that ease the migration of application GUIs have already been proposed [Fleurey et al., 2007, Sánchez Ramón et al., 2014, Robillard and Kutschera, 2019]. However, each proposed approach is dedicated to the migration of GUI using one specific framework. For companies using several GUI frameworks, it would impose re-developing a migration approach for each application. Moreover, these existing approaches only perform partial GUI migrations. For example, some approaches do not automatically migrate the possible interactions with the end-user. Thus, the remaining code must be manually migrated before the new system

[2] See https://en.wikipedia.org/wiki/List_of_widget_toolkits
[3] https://angular.io/
[4] https://reactjs.org/
[5] https://vuejs.org/
[6] https://emberjs.com/

is delivered. Such a manual step could take months when migrating large applications. Consequently, the development of new functionalities would be blocked during this period, and the final users would not receive any update. Such an approach is not acceptable in an industrial context with intense pressure to deliver.

1.2 Problem

The industrial context in which this thesis takes place allows us to identify two problems when it comes to supporting GUI migration: *support GUI frameworks agnostic migration*, and *enable incremental migration*.

Support GUI frameworks agnostic migration. Tools and approaches have been proposed to support GUI migration [Fleurey et al., 2007, Włodarski et al., 2019, Joorabchi and Mesbah, 2012, Samir et al., 2007, Shah and Tilevich, 2011]. They use meta-models to represent the GUI structure. However, these approaches are not sufficient for modern applications. Indeed, the authors did not apply their migration approaches to multiple and different GUI frameworks. Moreover, the authors present only partially their GUI models which makes it hard to adapt their solutions to other contexts.

Enable incremental migration. The existing approaches ease the migration by performing a partial GUI migration. Whereas they give good results automatically migrating an important part of application GUIs, developers must manually fix the remaining code before delivering the new system to customers. During the manual step, the old version of the software system is not maintained. Thus, the customer can not benefit from bug fixes and improvements until the end of the migration process. This can take months. Moreover, systems must be updated according to each country laws that evolve independently of the company migration project. Thus, not maintaining the former software system is not feasible.

One solution is to migrate the applications incrementally. Robillard and Kutschera [2019] used a hybrid architecture to mix Java Swing and JavaFX. This allows one to migrate part of the application while maintaining the part not migrated. Whereas this solution offers GUI hybridization, it does not fit in web contexts used in many industrial projects. Teppe [2009] proposed to migrate a part of an application with an automatic tool, test the migrated part, improve the automatic tool in case of issues, and iterate. When all the parts are migrated, they switch from the source application to the migrated one. However, all along the migration process, when new developments are made in the source application, developers must replicate the modification in the migrated version. Thus, developers have to ensure the maintenance of two applications during the migration process.

1.3 Our Approach in a Nutshell

To tackle the above problems, we designed an approach and meta-model that ease GUI migration. The approach is summarized in Figure 1.1. It takes an application GUI as input and migrates the GUI incrementally.

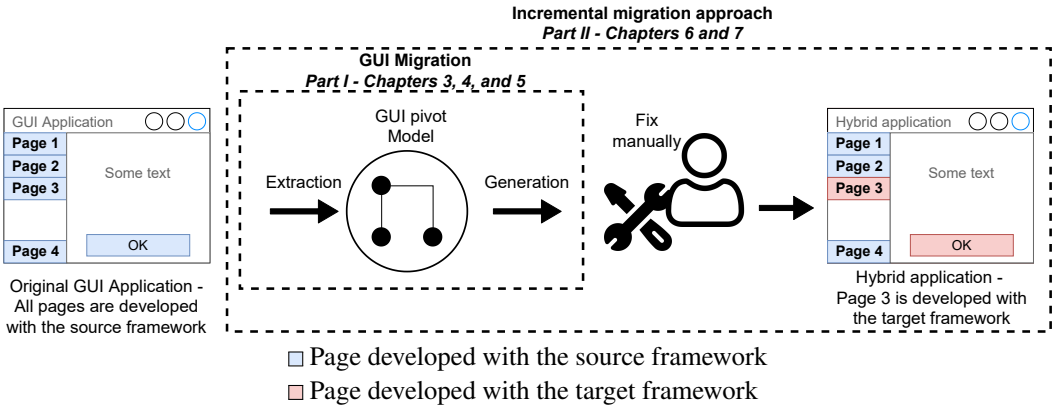


Figure 1.1: Summary of the approach

To tackle the first problem, *support GUI frameworks agnostic migration*, we proposed a highly extensible GUI migration approach with its meta-model. It is the central part of Figure 1.1. The approach comes with:

- steps that one can adapt to extract and generate GUIs in other contexts (*i.e.* other frameworks);
- extensible meta-model to represent the GUI to migrate; and
- implementation examples that extract GUIs from applications using a web-based framework (Java/GWT) or desktop-based framework (Pharo/Spec) and to generate applications using a web-based framework (TypeScript/Angular and Pharo/Seaside), or desktop-based framework (Pharo/Spec2).

To tackle the second problem, *enable incremental migration*, we built a hybrid architecture that authorizes one to mix different GUI frameworks (right-hand side of Figure 1.1) in the same application. Our hybrid architecture is part of an incremental approach that migrates web applications part by part (*i.e.*, page by page or module by module).

We implemented the hybrid architecture and used it during an industrial migration project from GWT to Angular. The results show that our hybrid architecture enables the migration of the application. It allows one to mix GWT and Angular without performance losses.

In summary, our approach consists of taking an application with its GUI as input, automatically migrating the application part by part using a tool based on a GUI meta-model, fixing each part manually and integrating them in a hybrid application. The hybrid application is then delivered to end users all along the migration process.

1.4 Contributions

The main contribution of this thesis is an incremental GUI migration approach. It includes:

- A detailed approach to migrate application GUI;
- A meta-model representing the GUIs;
- A hybrid architecture enabling incremental migration.

1.5 Structure of the Thesis

We organized the thesis as follows: Chapter 2 presents the state of the art related to this thesis. We list the relevant literature and highlight the shortcoming of existing solutions. Then, the thesis is split into two parts. First, in Part I, we present our solution to support GUI migration:

- Chapter 3 details the approach we designed to support the GUI migration.
- Chapter 4 presents several implementation details that migrate different application GUIs.
- Chapter 5 validates our approach and meta-model on five migration projects including real case app.

Then, in Part II, we present how we enable the migration of large industrial applications:

- Chapter 6 presents how we enable incremental GUI web application migration using a hybrid architecture.
- Chapter 7 validates the hybrid architecture and the incremental approach on an actual industrial migration.

Finally, Chapter 8 summarizes and concludes the work presented in this thesis and proposes future work.

1.6 List of Publications

The list of papers published in the context of the thesis is listed below in chronological order:

1. Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Abderrahmane Seriai, Laurent Deruelle, Stéphane Ducasse, and Mustapha Derras. GUI migration using MDE from GWT to Angular 6: An industrial case. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, pages 579–583, Hangzhou, China, 2019b. doi: 10.1109/SANER.2019.8667989. URL https://hal.inria.fr/hal-02019015

2. Benoît Verhaeghe, Anne Etien, Stéphane Ducasse, Abderrahmane Seriai, Laurent Deruelle, and Mustapha Derras. Migration de GWT vers Angular 6 en utilisant l'IDM. In *Conférence en Ingénierie du Logiciel*, Toulouse, France, June 2019c. URL https://hal.inria.fr/hal-02304296

3. Benoît Verhaeghe, Nicolas Anquetil, Stéphane Ducasse, Abderrahmane Seriai, Laurent Deruelle, and Mustapha Derras. Migrating GWT to Angular 6 using MDE. In *12th Seminar on Advanced Techniques & Tools for Software Evolution*, Bolzano, Italy, July 2019a. URL https://hal.inria.fr/hal-02304301

4. Clement Dutriez, Benoît Verhaeghe, and Mustapha Derras. Switching of GUI framework: the case from Spec to Spec 2. In *Proceedings of the 14th Edition of the International Workshop on Smalltalk Technologies*, Cologne, Germany, August 2019. URL https://hal.archives-ouvertes.fr/hal-02297858

5. Santiago Bragagnolo, Benoît Verhaeghe, Abderrahmane Seriai, Mustapha Derras, and Anne Etien. Challenges for layout validation: Lessons learned. In *International Conference on the Quality of Information and Communications Technology, QUATIC'2020*, September 2020b. URL https://hal.inria.fr/hal-02914750

6. Benoît Verhaeghe, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, Abderrahmane Seriai, and Mustapha Derras. GUI visual aspect migration: a framework agnostic solution. *Automated Software Engineering*, 28(2):6, 2021a. ISSN 0928-8910. doi: 10.1007/s10515-021-00284-z

7. Benoît Verhaeghe, Anas Shatnawi, Abderrahmane Seriai, Anne Etien, Nicolas Anquetil, Mustapha Derras, and Stéphane Ducasse. From GWT to Angular: An experiment report on migrating a legacy web application. *IEEE Software*, 2021c

8. Benoît Verhaeghe, Nicolas Anquetil, Anne Etien, Abderrahmane Seriai, Anas Shatnawi, Stéphane Ducasse, and Mustapha Derras. Migrating GUI behavior: from GWT to Angular. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'21)*, Luxembourg City, Luxembourg, September 2021b

9. Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Abderrahmane Seriai, Anas Shatnawi, Mustapha Derras, and Stéphane Ducasse. An hybrid architecture for the incremental migration of web front-end. In *International Conference on Advanced Information Systems Engineering (CAiSE'22)*, Leuven, Belgium, June 2022. (in submission)

We also participated in other research topics during the thesis. Since these topics are not directly relative to the focus of this thesis, we included the list of the corresponding papers below:

1. Benoît Verhaeghe, Nicolas Anquetil, Stéphane Ducasse, and Vincent Blondeau. Usage of tests in an open-source community. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, IWST '17, pages 4:1–4:9, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5554-4. doi: 10.1145/3139903.3139909

2. Serge Demeyer, Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, and Stéphane Ducasse. Evaluating the efficiency of continuous testing during test-driven development. In *Proceedings VST 2018 (2nd IEEE International Workshop on Validation, Analysis and Evolution of Software Tests)*, pages 1 – 5, March 2018. URL https://hal.inria.fr/hal-01717343

3. Benoît Verhaeghe, Christopher Fuhrman, Latifa Guerrouj, Nicolas Anquetil, and Stéphane Ducasse. Empirical study of programming to an interface. In *Proceedings of 34th Conference on Automated Software Engineering (ASE'19)*, San Diego, United States, November 2019d. doi: 10.1109/ASE.2019.00083. URL https://hal.inria.fr/hal-02353681

4. Nicolas Anquetil, Anne Etien, Mahugnon Honoré Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatija Djareddir, Jèrome Sudich, and Mustapha Derras. Modular moose: A new generation of software reengineering platform. In *International Conference on Software and Systems Reuse (ICSR'20)*, number 12541 in LNCS, December 2020

CHAPTER 2

State of the Art

Contents

2.1 GUI representation 9
2.2 GUI migration 13
2.3 Incremental migration 18

In Chapter 1, we identified two main challenges when migrating applications: (1) *migrating GUI using several GUI frameworks* and (2) *enabling end-to-end incremental migration*.

In this chapter, we review the scientific literature to assess the state of the art related to these problems. We first present the GUI representations proposed in the literature and the existing GUIs migration approaches in Section 2.1 and Section 2.2. Then, we detail the challenges and their existing solution to migrate GUI incrementally in Section 2.3.

2.1 GUI representation

Hayakawa et al. [2012] divided the user interface into four parts: *Meta* that contains information such as the title of the UI, *Widget* that includes the differents widgets types, *Style* that corresponds to CSS information, and *Behavior* that contains the code executed on user interactions. Following the proposition of other existing approaches [Fleurey et al., 2007, Garcés et al., 2017, Sánchez Ramón et al., 2014, Samir et al., 2007], we present all the visual aspect (*Meta*, *Widget*, *Style*) together in Section 2.1.1.

As depicted by Rodríguez-Echeverría et al. [2011] and Sánchez Ramón et al. [2016], the GUI layout must be represented in addition to the visual aspect. Thus, we present the layout representation in Section 2.1.2.

Note that, to the best of our knowledge, there is no detailed representation of the *Behavior* part described by Hayakawa et al. [2012] in current published work. Thus, it will not be part of this state of the art and will be discussed in the following chapter.

2.1.1 Visual part meta-model

To represent applications GUIs, some authors propose to use meta-models. These meta-models represent the GUI in the context of the authors' work. In the following, we discuss the proposed GUI meta-models.

2.1.1.1 OMG standards

The Object Management Group (OMG) defines two meta-models to represent the GUI visual aspect. The Knowledge Discovery Metamodel (KDM) allows one to represent any applications. The Interaction Flow Modeling Language (IFML) is specialized in applications with a GUI.

The KDM standard defined a meta-model to represent a piece of software at a high level of abstraction. It includes a UI package representing the elements of a GUI.

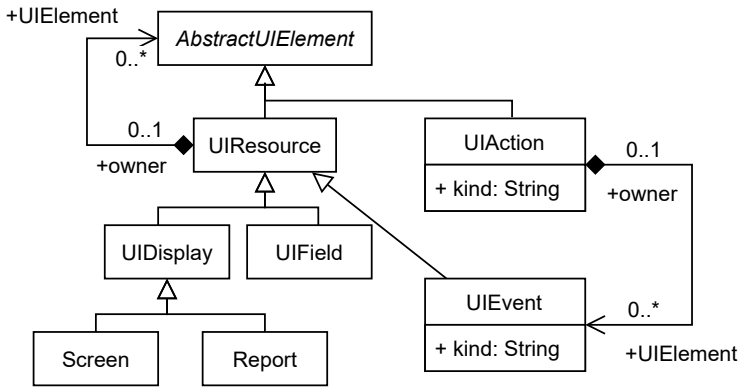


Figure 2.1: KDM - UIResources Class Diagram

Figure 2.1 represents the core of the UI package called UIResources Class Diagram. The central entity is UIResource. It can be refined as UIDisplay or UIField. UIDisplay corresponds to the physical support on which the interface will be displayed, *e.g.*, a computer screen, a printed report. UIField corresponds to a user interface widget. The composition between UIResource and AbstractUIElement is used to define the Document Object Model (DOM). Each UIResource can contain another one to represent a widget that contains other widgets.

Any UIResource can have a UIAction representing the user interface behavior. However, the behavior is not described in this meta-model.

The aim of IFML [Brambilla and Fraternali, 2014] is to describe an application GUI.

Figure 2.2 represents the IFML meta-model. The visible elements of the GUI

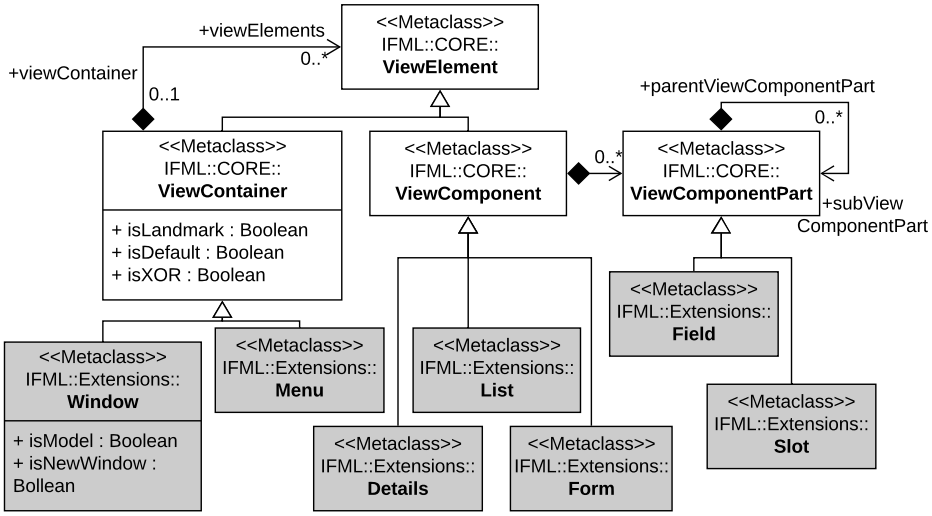


Figure 2.2: IFML - View Elements

are called ViewElement. A ViewElement can be refined as a ViewContainer or a ViewComponent.

A ViewContainer represents a container of other ViewContainer or ViewComponent. For example, it can be a window, an HTML page, a panel, *etc.* The composition between ViewContainer and ViewElement is used to define the DOM.

A ViewComponent corresponds to a widget that displays its content, *e.g.* a form, a data grid, an image gallery, *etc*. It can be linked to multiple ViewComponentPart. A ViewComponentPart represents an element of a ViewComponent. For example, an input field inside a form, a text displayed inside a data grid, or an image element of a gallery.

The two meta-models proposed by the OMG give a standard for further research. However, these meta-models need to be adapted to the different research contexts.

2.1.1.2 Adapted GUI meta-models

Whereas the OMG proposed a standard to represent GUIs applications, several GUI meta-models have been proposed in the literature. As proposed by the OMG, all researchers used a hierarchical representation of the GUI in the form of a DOM [Gotti and Mbarki, 2016, Fleurey et al., 2007, Mesbah et al., 2012, Memon et al.,

2003, Samir et al., 2007, Shah and Tilevich, 2011, Joorabchi and Mesbah, 2012, Brambilla and Fraternali, 2014]. Each node in the DOM tree represents a *widget* of the user interface. Thus, this representation is not controversial, and representing the DOM appears as a good solution to represent the GUI skeleton.

In addition to the DOM and the widgets, some authors added *attributes*. It is the case of Gotti and Mbarki [2016] who worked on analyzing the GUI structure of Java projects, Garcés et al. [2017] during the migration of oracle forms application, Memon et al. [2003] who extracted a GUI model from Java applications or MS Windows applications at runtime, Samir et al. [2007] during the migration process of Java application, Shah and Tilevich [2011] who extracted GUI to facilitate desktop to mobile migration, and Joorabchi and Mesbah [2012] who analyzed IOS applications. In the light of the diversity of work, representing the *attributes* is essential when modeling GUIs.

Others use the concept of *events* to later link the visual aspect with the GUI behavior. It is the case of Fleurey et al. [2007] for the migration of an industrial application, Mesbah et al. [2012] for the analysis of Ajax-based application and Garcés et al. [2017], Samir et al. [2007], and Joorabchi and Mesbah [2012] again. Representing *events* is a common task when migrating applications. In their work, the behavior of an application is only about navigation between pages.

Gotti and Mbarki [2016] and Sánchez Ramón et al. [2016] propose to specialize their meta-models with different kinds of known widgets such as Button, Label, Panel, *etc*. Thus, instead of using a generic widget concept, they can map each widget of the source code to its counterpart in their GUI meta-model.

Finally, Sánchez Ramón et al. [2016] also define a special kind of widget named "custom" to support widgets that their GUI extraction approach might not recognize.

All researchers use the DOM to represent the visual part with the attribute and the event concepts. The authors also proposed modifications to adapt their meta-models to their specific context. However, no study presents how those meta-models can be used in other contexts. For instance, there is no proposed adaptation regarding the use of a meta-model representing desktop application GUI to represent a web application GUI. Moreover, only Sánchez Ramón et al. [2016] present the concept of a custom widget to deal with unknown widgets. Thus, one needs to define an extensible GUI meta-model to reuse it in different contexts.

2.1.2 Layout meta-models

To correctly represent the visual aspect of a GUI, a layout representation is also necessary [Rodríguez-Echeverría et al., 2011, Sánchez Ramón et al., 2016]. There are three identified layout managers in the literature: hardcoded, hierarchical, and

constraint-based:

- **Hardcoded layout** defines for each widget its position with absolute coordinates on the screen [Sánchez Ramón et al., 2016]. It is used in old GUI frameworks. This layout is less and less used since the apparition of the DOM representation.
- **Hierarchical layout** consists of subdividing the available space of the screen into panels. Then the panels are responsible for placing their children in the dedicated space [Hasselknippe and Li, 2017]. Sánchez Ramón et al. [2014] propose a layout meta-model that supports hierarchical layouts. Zeidler et al. [2012] claim that the grid-bag layout, which is a hierarchical layout, is the most prominent and that almost all available GUI frameworks support it. Meliá et al. [2008] work on Model-Driven Development for GWT applications. In GWT, a widget is at the same time a visual element and a layout. So, they designed a meta-model with GUI components in which the container elements are at the same time container and hierarchical layout manager, for example, the GridPanel, the HorizontalPanel, and the VerticalPanel.
- **Constraint-based layout** also uses a hierarchical structure, but it uses constraints to place the widgets, for example: "place this button on the right of this text". Lutteroth et al. [2008] presented the Auckland Layout Model, which is an implementation of a constraint-based layout.

Authors have proposed layout meta-models to represent GUIs accurately. Several representations exist depending on the research contexts. The hierarchical layout comes out as the most used layout manager.

2.2 GUI migration

Using their GUI representations, the authors have proposed migration approaches.

We will first rapidly mention recent work on GUI generation using Artificial Intelligence (from screenshot examples). It is the case of Beltramelli [2017], Chen et al. [2018], and Moran et al. [2018]. These approaches rely on a huge dataset of screenshot examples (14,382 screenshots for Moran et al. [2018] and 10,804 for Chen et al. [2018]) to train the model. Thus, Beltramelli [2017] warns that the approach "is not, in any way, intended, nor able to generate code in a real-world context" and "both the source code and the datasets are provided to foster future research [...] and are not designed for end-users"[1]. Consequently, we rule out Artificial Intelligence as a possible approach given the current state of the art.

[1] https://github.com/tonybeltramelli/pix2code#disclaimer

We identified various publications related to migration. Section 2.2.1 presents generic approaches to migrate applications. Section 2.2.2 focuses on the language conversion research field. Section 2.2.3 presents existing projects migrating the visual aspect. Section 2.2.4 details the steps proposed in the literature to extract the GUI representation.

2.2.1 Migration approaches

Sneed and Verhoef [2020a] described three ways to migrate an application: *conversion*, *reimplementation*, and *wrapping*.

Conversion consists in a one-step approach that translates statement by statement the source code to its target language counterpart [Brant et al., 2010].

Reimplementation is used by many approaches that migrate the GUI visual code [Fleurey et al., 2007, Garcés et al., 2017, Sánchez Ramón et al., 2014, Hayakawa et al., 2012, Mesbah and van Deursen, 2007]. It follows this process:

1. The old application is extracted into a source language-specific model.
2. Then, the model is transformed to a higher-level representation.
3. Finally, the high-level model is transformed into a target language-specific model or directly used to generate the target application.

Wrapping "is an established re-engineering technique to provide access to existing functionality through a preferred interface" [Tonelli et al., 2010]. In consequence, the source code is not migrated but called by the new code.

Each approach allows one to execute code with the target GUI framework. However, only *conversion* and *reimplementation* perform a migration. Moreover, *reimplementation* is the most used one for GUI migration. Thus, it is the one we will focus on.

2.2.2 Language conversion

We first present *language conversion* related work. The language conversion field focuses on migrating applications written in one programming language to another programming language. Whereas this work does not deal directly with GUI behavior, it deals with the reimplementation of source code in another language, which is part of the GUI behavior migration.

Malton [2001] classifies language conversion according to their difficulties into three categories:

Dialect conversion deals with the migration from one version of a programming language to another. It is the case of Python 2 to Python 3 migration [Aggarwal et al., 2015].

API migration is the switch of frameworks while keeping the same programming language [Teyton et al., 2013], for example, moving from Java Swing to JavaFX.

Language migration deals with the migration from one language to another. It better fits our context. Thus, we focus on this category in the rest of this section.

Brant et al. [2010] migrate a Delphi application into C#. To do so, they developed and used SmaCC, a transformation engine that allows one to write transformation patterns.

Terwilliger et al. [2012] work on the conversion of Fortran to C++ code. To do so, they developed the FABLE tool that automatically rewrites the code in C++. The authors wanted to generate C++ code suitable for future development, and at the same time "similar to the original Fortran code".

Martin and Muller [2002] translate C code to Java. To translate the application, they used a traditional approach: create an AST representation of the source code, transform it into an AST for Java, and then traverse this AST to generate the target source code.

Trudel et al. [2012] migrate C to Eiffel (an object-oriented language). They developed a tool that builds an AST of the source code and applies successive transformations to this AST. Thus, the tool incrementally transforms the code from C to Eiffel. The authors also manually wrote helper classes that ensure the Eiffel classes have the same capabilities as their C structure counterparts. For example, there is a helper class to access the *stdio* library aiming to help Eiffel translated code using *stdio* specific features.

These approaches point out that an AST representation is necessary when migrating behavioral code. It must come with a parser to build the AST and transformation rules to perform the migration.

2.2.3 Existing migration project

We are interested in a generic GUI migration approach that handles multiple source and target frameworks. Thus, we are interested in whether the proposed solution can (i) import GUI from markup languages (*e.g.*, HTML), (ii) import GUI from programming languages (*e.g.*, Java Swing); (iii) import from binary source (*e.g.*, Oracle form); (iv) handle multiple source frameworks; (v) export GUI to markup

languages; (vi) export GUI to programming languages. We will not consider exporting to a binary framework as no modern GUI framework uses this approach anymore.

Table 2.1: Existing migration project

	extract markup language	extract programming language	extract binary	multiple frameworks	export markup language	export programming language
Our needs	✓	✓	✓	✓	✓	✓
Hayakawa et al. [2012]	✓			✓	✓	
Mesbah and van Deursen [2007]	✓				✓	
Bragagnolo et al. [2020a]			✓		✓	
Garcés et al. [2017]			✓		✓	
Sánchez Ramón et al. [2014]			✓			✓
Fleurey et al. [2007]		✓			✓	
Samir et al. [2007]		✓			✓	
Robillard and Kutschera [2019]		✓				✓

Prior research makes valuable contributions: migration "process" (see Section 2.2.4) and/or GUI internal representation (models, see Section 2.1.1). However, there are rarely enough details to generalize the approaches to other languages/frameworks.

Table 2.1 summarizes the related work considered.

Some past research considered migrating from markup languages: Hayakawa et al. [2012] (multiple markup languages), Mesbah and van Deursen [2007] (multi-page web application to Single Page Application (SPA) using Ajax). Migrating from markup languages is easier because the language is simple to parse (there are numerous parsers for HTML or XML), detecting the GUI elements (widgets) is straightforward (*e.g.*, a tag <button>), and the DOM clearly describes the structure of the interface.

Sánchez Ramón et al. [2014] and Garcés et al. [2017] considered the case of Oracle Forms, a framework that we classify as a binary source since there is no textual representation of the GUI (or an incomplete XML representation [Garcés et al., 2017, again]). Bragagnolo et al. [2020a] also worked on GUI extraction based on binary sources with the migration of Visual Basic applications. Sánchez Ramón et al. [2014] consider migrating to Java Swing (programming language), Garcés et al. [2017] to JEE application (markup language because the GUI is defined in HTML files), and Bragagnolo et al. [2020a] to Angular (markup language). The publications focus on the extraction part of the process since there are specific problems to access the GUI representation of binary frameworks. Their GUI meta-models are valuable (see Section 2.1.1) as well as their generic process (see 2.2.4), however, there are not enough details to generalize them to other languages.

Fleurey et al. [2007], Samir et al. [2007], and Robillard and Kutschera [2019] consider the extraction of GUI based on a programming language. The first one

from Coolgen generated code[2], the second and the third ones from Java Swing code. Even if none of the publications detail how to adapt the approach to extract GUIs based on other programming languages, they give hints on the general approach, such as how to map source and target widgets. The first two migrate to markup language based GUI: Fleurey et al. [2007] migrate to J2EE, and Samir et al. [2007] to Ajax Web with XUL[3]. The third work migrates to the JavaFX framework (programming language based GUI). We note that the XUL format has been discontinued.

Thus, none of the existing projects deal with the multi-framework migration defined with markup and programming languages. Furthermore, only Robillard and Kutschera [2019] present the migration from a GUI based on a programming language to another. However, in their context, the GUI is always defined using the Java programming language (from Java Swing framework to JavaFX framework), which has eased the migration.

2.2.4 GUI visual code extraction approaches

The reimplementation approaches often use the horseshoe process [Kazman et al., 1998]. To perform the extraction of their GUI models, authors follow similar steps adapted to their contexts. Note that they only detail the steps they used for the extraction depending on their work focus. In the following, we present these steps common to several approaches and explain their adaptation for markup and programing language found in the literature.

Map source to meta-model concepts. All authors use a dictionary that maps the widgets of the source framework to their meta-models. In the case of markup language, Hayakawa et al. [2012] map the XML tags name to the widget concepts. In the case of programming language, Samir et al. [2007] propose to map the programming language classes (*i.e.*, `JPanel` class for Java Swing) to their equivalent meta-model concepts.

In case of missing concept in the meta-model, Sánchez Ramón et al. [2016] propose to map the source element to a custom widget.

Identify containment. Another step is the identification of the containment tree. This step allows authors to extract the visual aspect of the GUI as a DOM (see Section 2.1.1). In the case of markup language, Memon et al. [2003] propose to use the existing DOM defined in the source language to extract easily a DOM representation.

Identify root widget. Additionally to the containment, one step is to identify the root widget of a GUI. The root widget is the widget that contains all the others

[2]Coolgen: https://en.wikipedia.org/wiki/CA_Gen
[3]https://developer.mozilla.org/en-US/docs/Archive/Mozilla/XUL

(*e.g.*, a window, a web browser). In the case of markup language, Mesbah and van Deursen [2007], and Memon et al. [2003] propose to look at the configuration file of the source project. In the case of programming language, Rodríguez-Echeverría et al. [2011] define a set of widgets that can be the GUI's root widgets.

Create widget instances. When extracting the Visual code, approaches create the widget instances of each GUI. While the *Map source to meta-model concepts.* step defines links between the source framework and the meta-model; this step aims to identify where each widget is used. In the case of programming language, Rodríguez-Echeverría et al. [2011] and Silva et al. [2010] propose to create the widget instances by looking at the widget instantiation in the source code. They precise that a widget instantiation comes from a call to its constructor (*e.g.* `new JButton()`) or from the usage of a widget factory.

Additional sub-step. Finally, Silva et al. [2010] propose an extra step for the programming language. It consists of using symbolic execution to resolve more precise information about widget position in the layout. To perform this kind of advanced extraction process, Deltombe et al. [2012] propose to use an AST meta-model in addition to a more global meta-model representing the source application. It allows them to navigate between the precise and the more abstract representation of an application.

Several steps already exist for the extraction of the GUI visual aspect. The authors have described how to perform the steps in their specific context. However, there is a lack of an overall approach that would help build new extractors for another GUI framework. Moreover, whereas there are steps described to extract the visual code, none of the published work discusses the code's generation in a target language.

2.3 Incremental migration

To perform an incremental GUI migration, the solution proposed in the literature is to use a hybrid application mixing both the source and target GUI. We identified various publications related to designing or using a hybrid architecture. Section 2.3.1 presents the challenges to build a hybrid architecture. Section 2.3.2 presents the existing hybrid architectures and the constraints they fulfill.

2.3.1 GUI migration constraints

Several constraints must be taken into consideration when designing a hybrid architecture. Terekhov and Verhoef [2000] and Chisnall [2013] detail the challenges to make two programming languages interoperable. In the following, we present the constraints raised in the literature.

Communication In the case of a hybrid application, several programming languages are involved in the implementation. At runtime, each language has to communicate with the others (*e.g.*, invoking methods from one programming language to another). Chisnall [2013] details the difficulties of bridging two programming languages. For instance, they report the need for C interfaces to enable communication between Java and C++.

Type matching One major challenge faced when designing a hybrid architecture is the matching of data types [Terekhov and Verhoef, 2000, Chisnall, 2013]. The two programming languages might have different structure representations for a type. For instance, Java primitive types are implemented using specific Java wrapping classes, whereas, in JavaScript, primitive types are common JavaScript types. Thus, a *number* in JavaScript can not be directly translated into a Java *Integer*.

GUI mixing The need to mix GUI has only been raised by Robillard and Kutschera [2019]. In such a context, widgets defined in different GUI frameworks are present within the same page. Thus, a strategy must be developed to enable the integration of one GUI with the other.

2.3.2 Hybrid architecture

In the following, we detail existing projects using a hybrid architecture and the constraints they deal with.

Robillard and Kutschera [2019] work on the migration of a Java Swing application to JavaFX. They migrated the application incrementally by mixing Java Swing and JavaFX components. They deal with the communication and GUI mixing constraints. In their context, dealing with these problems was eased because Java Swing and JavaFX are both developed in Java. Thus, communication between the two GUI frameworks is straightforward since both are implemented with the Java programing language. To mix GUIs, they used the existing GUI mixing capabilities of Java Swing and JavaFX.

Comella-Dorda et al. [2000] detail different strategies to mix application for modernization projects. For the user interface modernization, they propose to use screen scraping technic. It consists of analyzing the source application rendered UI at runtime, converting it, and wrapping it for the target platform (web-based or desktop-based). Flores-Ruiz et al. [2018] and Zhang et al. [2008] use this strategy with different implementations. Although this approach allows one to present the GUI of an application in another context (*e.g.* desktop-based GUI inside a web browser), it does not allow communication between the source GUI and the target one. Using screen scraping technic, the authors deal with the GUI mixing constraint but do not consider the communication between the hybridized elements.

Kontogiannis et al. [2010] propose a set of transformation rules to migrate from one programming language to another. Whereas they do not discuss how two different languages can communicate, their transformation rules deal with the type matching problem. To do so, they created a type correspondence table for Basic PL/IX to C. GUI mixing was out of their scope.

Teppe [2009] uses an iterative approach to migrate an SPL[4] application to C++. When performing the migration, he had to deal with the type matching problem. The author uses a type correspondence table and an intermediate layer that transforms a type defined in one language to its equivalent type in another language. This approach is only detailed for applications without GUI and running on the desktop (rather than the browser).

Finally, Sneed et al. [2006] propose to wrap legacy code to make it available as a web service. This strategy allows one to create communication between different programming languages. By making code available as a web service, the authors also deal with the type matching problem by serializing the data in XML format.

Additionally to the academic literature, a common way to mix web GUI is the usage of the *iframe* tag[5]. This solution allows one to insert inside one web page the content of another one. Whereas this solution is convenient and does not require designing a new architecture, it comes with substantial limitations. It is strongly discouraged to enable communication with the content of an *iframe* for security purposes, and the *iframe* content should not access the main page.

Table 2.2: Hybrid architectures fulfilling constraints

	Communication	**Type matching**	**GUI mixing**
Robillard and Kutschera [2019]	✓		✓
Comella-Dorda et al. [2000]			✓
Flores-Ruiz et al. [2018]			✓
Zhang et al. [2008]			✓
Kontogiannis et al. [2010]		✓	
Teppe [2009]		✓	
Sneed et al. [2006]	✓	✓	
Technical: iframe			✓

Table 2.2 summarizes the migration projects already designed. None of the existing approaches deal with the communication, type matching, and GUI mixing constraints. Thus, to enable the incremental migration of large applications, one needs to consider all these challenges and propose solutions.

[4] http://www.clifford.at/spl/

[5] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

We presented the current state of the art on the GUI migration. It consists of existing GUI meta-models dedicated to specific contexts as well as the extraction approaches to create models. We also presented the constraints to perform an incremental migration and the existing solution dealing with those constraints. However, none of the authors deal with all the constraints but only with specific ones. Thus, one needs to propose an incremental migration approach dealing with all the identified constraints, and that can be adapted to different contexts, *i.e.*, extracting GUI defined in markup and programming languages and exporting in markup and programming languages.

Part I

GUI migration

CHAPTER 3

Approach

Contents

3.1 GUI Terminology and Concept 25
3.2 Approach overview 31
3.3 Visual code migration approach 35
3.4 Behavioral code migration approach 46
3.5 Conclusion 53

In the literature, several approaches are proposed to migrate application GUIs. They point out two major challenges: designing GUI representations and using them for the migration.

To build a GUI representation, the first step is to define the different aspects composing a GUI. We define them in Section 3.1 and use them in the rest of this thesis. It includes a separation of the GUI into the visual, the behavioral, and the business aspects.

Then, we design a migration approach similar to the ones found in the literature including all GUI aspects (see Section 3.2). It includes the extraction of the application source code concepts to build a model that represents the visual aspect (see Section 3.3) and the behavioral aspect (see Section 3.4), and the generation of the target code. We detail the extraction and generation approach steps to ease future adaptation to other migration contexts.

3.1 GUI Terminology and Concept

As presented in the literature (see Section 2.1), there are several proposals to represent the GUI. Some authors proposed to represent only the widgets; others proposed to group the widgets and their attributes. Depending on their context, the authors also represent the GUI layout [Rodríguez-Echeverría et al., 2011, Sánchez Ramón et al., 2016] or the navigation between the pages [Joorabchi and Mesbah, 2012].

Based on the existing separation of the GUI code, we define three categories of source code: the Visual code, the Behavioral code, and the Business code.

Visual code The Visual code describes the visual aspect of the GUI. It contains the visual 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 relative to others (layout). We group the Visual code concepts and explained them in Section 3.1.1

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 actions are automatically triggered following an outside event. We detail the concept of Behavioral code in Section 3.1.2.

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. We did not deal with the automatic migration of Business code. However, we had to handle it in some way when performing an incremental migration (see Chapter 6). We present the Business code in Section 3.1.3.

3.1.1 Visual code

One part of a GUI is the Visual code. It corresponds to the elements used to create the GUI from the end-user point of view. We divided it into the widgets (see Section 3.1.1.1) that are the visible elements, and the layout (see Section 3.1.1.2) that defines the position of the widgets in the GUI.

3.1.1.1 Widget

A set of widgets and attributes commonly composes a user interface. The available widgets and attributes of the UI are defined in the GUI framework used by the developers.

Figure 3.1 presents a GUI example. All the visible elements are widgets.

Some widgets are simple, it is the case of the “Button Text” or the “Link”. They have a simple visual representation and few attributes. Some widgets have a complex visual representation. It is the case of the pie chart in the Figure 3.1. Others are container widgets. A container widget can include other widgets. It is the case of “Group A”, “Group B” and the toolbar at the top of the window in Figure 3.1.

Additionally to the standard widgets found in GUI frameworks, developers can create their custom widgets. To do so, they build a new component from scratch or from a combination of preexisting widgets.

The widgets have attributes that configure their visual aspect. For example, attributes set the text value of the button and link. They can also change the color of an element. It is the case for the color of the “Button Text”. Finally, they can

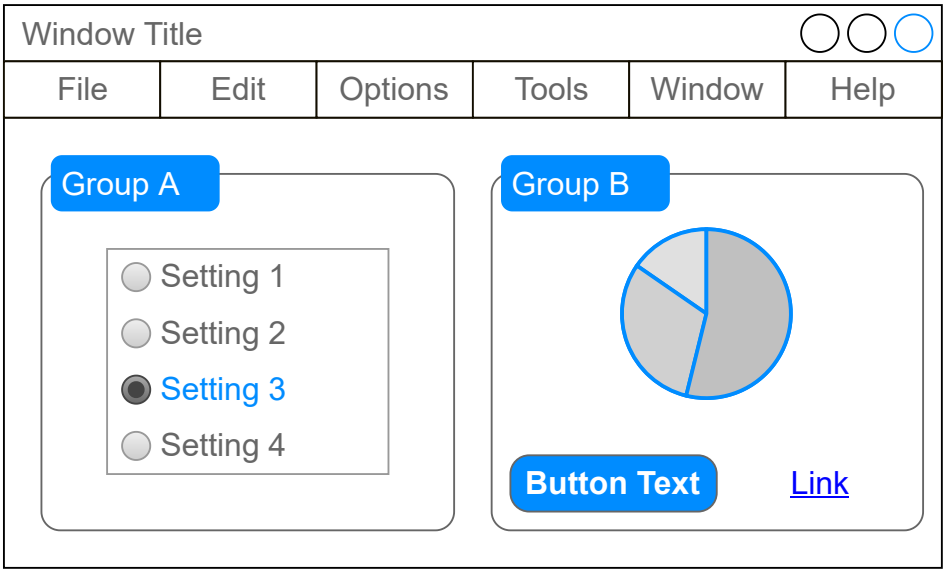


Figure 3.1: GUI example

link together different widgets. For instance, in "Group A", only one of the four radio buttons (*e.g.,* "Setting 1", "Setting 2", "Setting 3", and "Setting 4") can be selected at a time. To represent the relationship between the radio buttons, some GUI frameworks use a common attribute, *e.g.*, the `name` attribute with HTML, other GUI frameworks use a common widget container, *e.g.*, a `ToggleGroup` with JavaFX.

In summary, the widgets are the visible elements of a user interface, and the attributes configure the widgets (*e.g.*, text, color, . . .).
The position of the widgets in the GUI, for example, the vertical distribution of the four radio buttons and setting labels, or the horizontal distribution of the two groups A and B, is the layout responsibility.

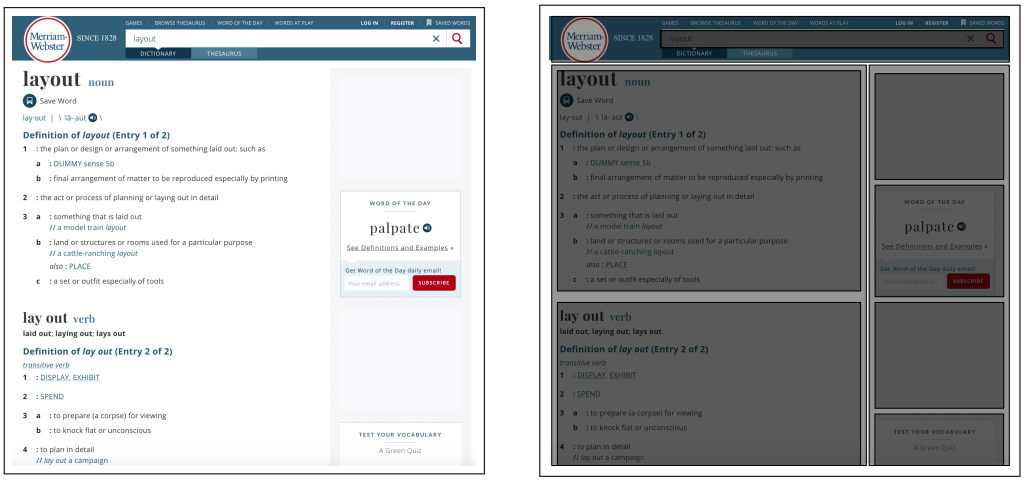
3.1.1.2 Layout

From Merriam&Webster dictionary[1] a Layout is *the plan or design or arrangement of something laid out.*

From this definition and the proposed layout representations of the literature (see Section 2.1.2, [Rodríguez-Echeverría et al., 2011, Sánchez Ramón et al., 2016]), we consider that the layout defines the widgets' position relative to others. There are two main kinds of components in a GUI: the containers that contain other components for defining groups of components and the leaves that are contained. The

[1] https://www.merriam-webster.com/dictionary/layout

containers, such as fieldsets, panels, *etc.*, are responsible for defining the disposition of the contained elements in the page.



(a) Page (b) Page's highlighted layout

Figure 3.2: A layout example

In Figure 3.2, we present the relationship between a GUI and its layout for the Merriam&Webster web page. Figure 3.2a shows the page as interpreted by the browser, and, in Figure 3.2b we revealed the layout.

Together, the widgets and the layout make up the visual code of a GUI.

3.1.2 Behavioral code

The Behavioral code is not as well described in the literature as the Visual code. Only Hayakawa et al. [2012] present it as "the executed script when an event is fired".

Thus, we propose a definition of what the Behavioral code is. To do so, we first present a concrete example in Section 3.1.2.1. Then, we split the Behavioral code into two parts and present them in Section 3.1.2.2.

3.1.2.1 A concrete example

To clarify the definition of Behavioral code, we use the following concrete example.

Listing 3.1 shows an example of Java code. It corresponds to a method executed when the end-user clicks on a button. The method reads the value of an inputText (line 3) looking for email addresses separated by commas (line 5) and uses a service to send an email to each address (line 6). In the following, we highlighted the code expressions that are part of the Behavioral code. These expressions are underlined in Listing 3.1.

```
1  button.addClickHandler(new ClickHandler() {
2    public void onClick(final ClickEvent event) {
3      String values = emailBox.getText();
4      if (values != null) {
5        List<String> results = values.split(",");
6        IService.sendEmail(results, new AsyncCallback<
     List<String>>() {
7          public void onSuccess(List<String> result){
8           EventPopup.displayInfo(result.toString());
9          }
10       });
11 }}});
12
```

Listing 3.1: Behavioral code in Java

First, line 1, addClickHandler(new ClickHandler ...) corresponds to the creation of an event click handler and attaches it to the widget button. The event is the entry-point of the Behavioral code. When the click event is fired, the method onClick line 2 is executed.

Then, on line 3, there are two behavioral elements. emailBox is an access to a UI element, here an input text of the UI. And .getText() is an access to the value of the attribute *text* of the widget emailBox.

Finally, on line 8, there is a declaration and usage of a popup window. The Popup is identified by the usage of the class EventPopup, then the type of the Popup (info, warning, error) is defined by the invoked method, *i.e.*, displayInfo line 8.

Except for the event handler that is the starting point of the Behavioral code, the other behavioral elements are code expressions that manipulate GUI data (*e.g.*, widget, widget's attributes, *etc.*). All other parts of the code, not directly linked to the UI, do not belong to the Behavioral code but to Business code (see Section 3.1.3). They are control flow (*if*, line 4) and algorithm details (converting a String as a List, line 5, or call to a distant service, line 6).

3.1.2.2 Behavioral code structure

From the previous example, we subdivide the Behavioral code into two categories: the events and the Manipulation code.

Events correspond to the events raised by the system or when end-users interact with the UI. Each GUI framework has a set of recognized events; however, there are some common ones, and each web browser proposes an exhaustive

events list[2].

Manipulation code impacts or references part of the visual aspect of the application. Examples of Manipulation code include showing or hiding widgets.

For the **Events**, by analyzing the applications of our industrial partner, we identified the following common events:

- *Click* corresponds to a user clicking on any UI element of the DOM. It can be a button as well as a table, a text, or an empty zone.
- *Change* is raised when end-users modify the content of a text input or table.
- *Error* corresponds to a problem, for example, when trying to load an image but the resource is unavailable.
- *Submit* corresponds to a user submitting a form.
- *SubmitComplete* is raised by the system after a successful *Submit* event, *e.g.*, when a user correctly filled form fields and no network problem happens.

For **Manipulation code**, there is no exhaustive list of possible expressions that impact the UI. So, we propose a first list of Manipulation code found in our context.

- *Widget access* is the reference in the code to a widget. For example, Listing 3.1 line 3: `emailBox`.
- *Widget attribute getter or setter* is the call to a widget attribute accessor. For example, Listing 3.1 line 3: `getText()`.
- *Navigating* corresponds to the navigation from one page to another.
- *Open Popup* shows a Popup in the application. Popups can be: *info*; *warning*; or *error*. For example, Listing 3.1 line 7: `EventPopup.displayInfo`.
- *Open Dialog*[3] is the piece of code used to open a dialog in the GUI. The visual aspect of the Dialog is defined in the Visual code of the application.

Again, other kinds of Manipulation code may exist, *e.g.*, adding or removing a widget of the GUI or widgets' animations. However, they do not exist in our context, so we did not consider them in the following.

[2]For example, for Firefox: https://developer.mozilla.org/en-US/docs/Web/Events

[3]A Dialog is a window "box or other interactive components, such as a dismissable alert, inspector, or subwindow" (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dialog)

3.1.3 Business code

The Business code consists of the code that does not deal with the visual and the behavioral aspect of the application. In that sense, it includes the application specific code such as the data structure used in the application. In the case of web applications, it includes the relationship between the front-end and the back-end.

In Listing 3.1 (page 29), the data manipulated by the Business code is hold by the values and the results variables. It is then sent to the service "*IService*".

From this snippet of code, we identify four Business code lines. First, line 3, the value of emailBox.getText is set in the values variable. Second, line 4, the value of the values variable is tested. Third, line 5, the data held by the values variable is split. Fourth, line 6, the data are sent to a remote service named IService.

Extracting the Business code of an application is a complex task and requires extensive work [Cariou et al., 2018, Sneed and Verhoef, 2020b]. We did not deal with the automatic migration of the Business code. However, we discuss it when performing the incremental migration of an application in Chapter 6.

3.2 Approach overview

Based on our GUI separation, we designed an approach to migrate applications' GUI. We first illustrate the approach with a concrete example demonstrating its aim in Section 3.2.1. Then, we present the approach and detail its steps in Section 3.2.3.

3.2.1 Concrete migration example

To clarify our approach, we first present a concrete example of migration. It follows the *Reimplementation* approach detailed in Section 2.2.1. It consists of the migration of a button defined in Java Swing to Angular.

The approach is divided into two main parts: extraction and generation (see Figure 3.3). The extraction takes the source code as input and builds a model representing the GUI. This model is then used as a pivot for the migration. In the following, we designate as pivot the elements of our abstract representation, allowing the migration of several GUI defined with different GUI frameworks. The generation takes the pivot model as input and produces the GUI target code.

The pivot model defines concepts that are common to all GUI migration projects and enables the migration of multiple GUI frameworks. The extraction (left-hand side of Figure 3.3) consists of building from the source code its corresponding pivot model. To do so, we first extract, from the code, instances of source language concepts in a first model. In the example, we extract the instantiation of a Java JButton from the new JButton("OK") piece of code. Then, the pivot model is built from

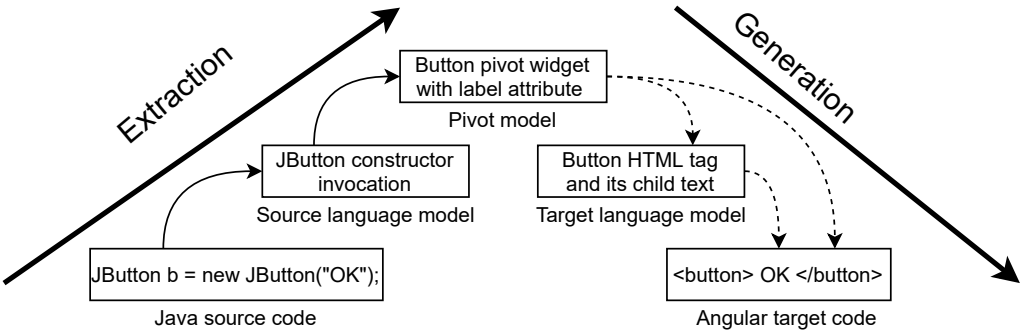


Figure 3.3: GUI migration concrete example from Java to Angular

the language model following some mapping rules. In the example, the constructor invocation is mapped to a button widget with its label attribute.

The generation (right-hand side of Figure 3.3) consists of the creation of the target code from the pivot model. First, the generic pivot model is transformed into a target language model. In the example, the pivot button is mapped to the concrete HTML tag <button> with a child HTML text to represent the label attribute. Then, we generate the Angular target code from the target language model.

Note that, as presented in Section 3.3.3, instead of concretely implementing the target language model, one can define the mapping from the pivot meta-model concepts directly to the target language code (dashed arrows in the figure). In such a case, the generation consists of one step from the pivot model to the target code. One can think of it as the target language meta-model being embedded in the concept mapping. We made this choice when designing our approach (see Section 3.2.3) and implementing it for several migration projects (see Chapter 4).

3.2.2 Pivot meta-model

In order to be independent of the source and target languages, markup or programming languages, our approach uses a pivot meta-model to represent the GUI. This meta-model aims to represent all the GUI concepts presented in Section 3.1. It is used as a pivot during the migration, *i.e.*, it is the target of the code extraction and the source of the code generation.

Our Pivot meta-model is split into four packages: core, widget, layout, and behavioral. The core, widget, and layout packages represent the Visual code of the GUI, and the behavioral package is dedicated to the Behavioral code. Figure 3.4 illustrates the different packages in the pivot meta-model.

Core package. The core package enables to represent the DOM of the GUI with an abstract representation of the widgets, their attributes, and the containment

relationship of the widgets ones with the others.

Widget package. The widget package adds to the core package several widget types.

Layout package. The layout package is dedicated to the representation of widgets' positions on the GUI.

Behavioral package. The behavioral package represents the GUI behavior when end-users interact with the visual components.

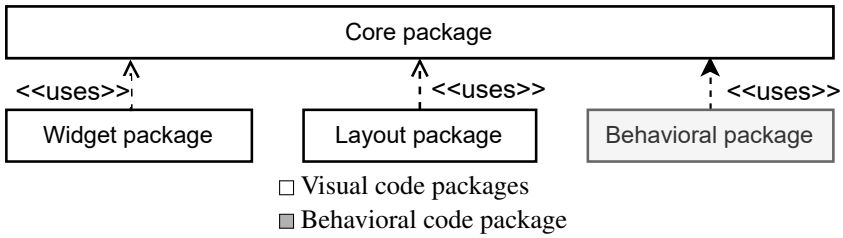


Figure 3.4: Pivot meta-model

This Pivot meta-model is used by our GUI migration approach. We detail it for the Visual code in Section 3.3.1 and for the Behavioral code in Section 3.4.1.

3.2.3 GUI migration approach

Based on our Pivot meta-model, we designed a migration approach in five steps. This approach is based on the concrete example illustrated in Figure 3.3. It includes the migration of the Visual code and the Behavioral code. Each step is divided into tunable sub-steps to enable multi-framework support. Examples of sub-step adjustments are illustrated with concrete cases in Chapter 4.

The process, represented in Figure 3.5, presents the steps to migrate the GUI. The step in white is the preliminary step that extracts a model of the source application; the steps in yellow correspond to the Visual code migration; and the steps in gray are used for the Behavioral code migration. Note that the migration of Behavioral code is based on the migration of Visual code, and both are based on the model extraction of the source application. The process is divided into the five following steps:

Source code model extraction. We build a model that represents the source code of the source application. To do so, one needs a source language parser and its meta-model. The source language can be a programming language

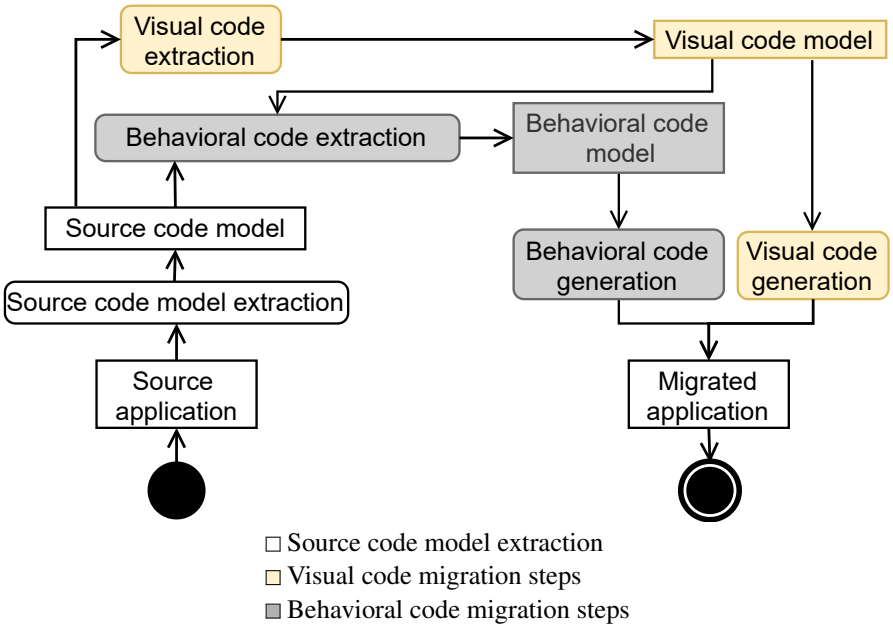


Figure 3.5: Our GUI migration process

or a markup language like XML or HTML. One can also extract a model from a binary file format. An example of a source code meta-model for programming languages is Famix [Ducasse et al., 2011].

Visual code extraction. We analyze the source code model to identify the visual elements. We build a mapping between the widgets of the source framework and the widgets of the GUI meta-model. Finally, we create a Visual code model (pivot model, with widget and layout parts) from this mapping.

Behavioral code extraction. This step takes as input the extracted Visual code model and the source code model. The source code model includes the Manipulation code and the creation of event handlers. The Visual code model includes the widgets already extracted with which users can interact. It produces a Behavioral code model (pivot model, with behavioral part).

Visual code generation. We re-create the visual aspect in the target language. First, we define the required configuration files and the target framework file architecture. Then, we define a mapping between the pivot meta-model concepts and their implementations in the target framework. Finally, we export the code corresponding to the visual part in the target language. We show examples of generators in Section 4.2.

Behavioral code generation. We export the extracted Behavioral code in the target

language inside the generated GUI code.

Our approach is similar to the three steps approach used in the literature (see Section 2.2.1). However, we added the support of the Behavioral code. The *Source code model extraction* step extracts the old application into a source language model. *Visual code extraction* and *Behavioral code extraction* steps transform the source model into a higher-level representation. It corresponds to the "Extraction" part of our concrete example Figure 3.3. They build a pivot model. The *Visual code generation* and *Behavioral code generation* steps transform the pivot model into the target application. It corresponds to the "Generation" part of our concrete example Figure 3.3. Note that, as detailed in Section 3.2.1, we did not implement the target language meta-model. Using a target language meta-model can be done through adding a step before the Behavioral code and Visual code generation that build the target language model from our GUI pivot model.

This approach presents the overall direction we follow to migrate an application. Each step is adapted to the specific migration contexts. In the following, we detail the steps for the Visual code migration (in yellow in Figure 3.5) and the steps for the Behavioral code migration (in gray in Figure 3.5).

3.3 Visual code migration approach

We presented our overall approach to migrate an application. We now detail the extraction and generation steps of the visual code part of the GUI migration. First, we design three packages part of the pivot meta-model that allow one to represent the Visual code (see Section 3.3.1). Then, we detail how to extract a model (see Section 3.3.2). Finally, we present the generation of target code from a model (see Section 3.3.3).

3.3.1 Visual code packages

We represent the Visual code thanks to three meta-model packages inspired by the literature (see Section 2.1.1). The core package includes the main GUI elements. The widget package adds existing components to improve the reusability of our approach. The layout package represents the widget GUI arrangement.

3.3.1.1 Core package

To represent the user interfaces of desktop or web-based applications, we designed the core package presented in Figure 3.6. The core represents the DOM of a user interface. Developers can then tune the meta-model by adding new entities to fit

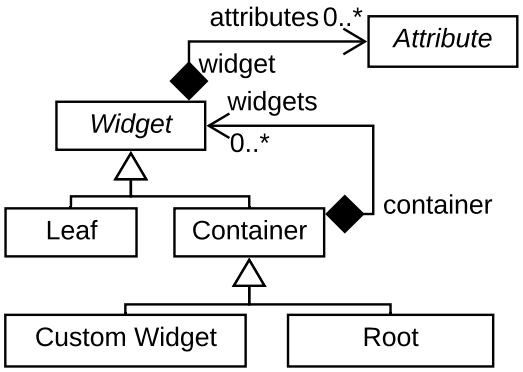


Figure 3.6: Core package

their requirements, such as additional widgets, for example, a complex parameterizable table component. For instance, to increase the reusability of our approach, we extended the core meta-model with several widgets and attributes in the widget package (see Section 3.3.1.2).

`Widget` is a graphical resource. It can be refined as `Leaf` or `Container`.

`Container` is a composite of `Widgets`.

`Leaf` is a basic widget that can not contain another widget. For example, it is the case of text input.

`Root` represents the main container of a graphical interface. It is either a window of a desktop application or a web page. The `Root` is a kind of `Container`.

`Attribute` represents a widget property. For example, a *button* may have a *text* attribute. An attribute can also change the behavior of a widget. It is the case of the attribute *enable*. A button with the *enable* attribute set to *false* represents a button on which one can not click.

`Custom Widgets` is a kind of `Container` that represents an unknown widget in our meta-model. During the migration process, it represents a detected but not recognized widget. The `Custom Widgets` concept is further discussed Section 3.3.1.4.

The DOM, massively used in the literature (see Section 2.1.1), is represented with the relation between `Container` and `Widget`. To represent the widget visual disposition, we introduced a layout package (see Section 3.3.1.3) representing the DOM with additional information such as how children are visually disposed inside their parent.

3.3.1.2 Widget package

The core package allows one to represent the GUI structure. However, using only the core package would not be sufficient to perform the migration. Indeed, the `Widget` concept should be refined to represent the diversity of existing widgets [Gotti and Mbarki, 2016, Sánchez Ramón et al., 2016]. It is the goal of the widget package.

The widget package describes the most common user interface widgets. It currently contains all the entities described in the W3School website[4] such as Button, Label, or Table. Note that this website only presents the widgets of the HTML standard. Our widget package is composed of 61 widgets and 31 attributes. An excerpt[5] of the widget package is presented in Figure 3.7.

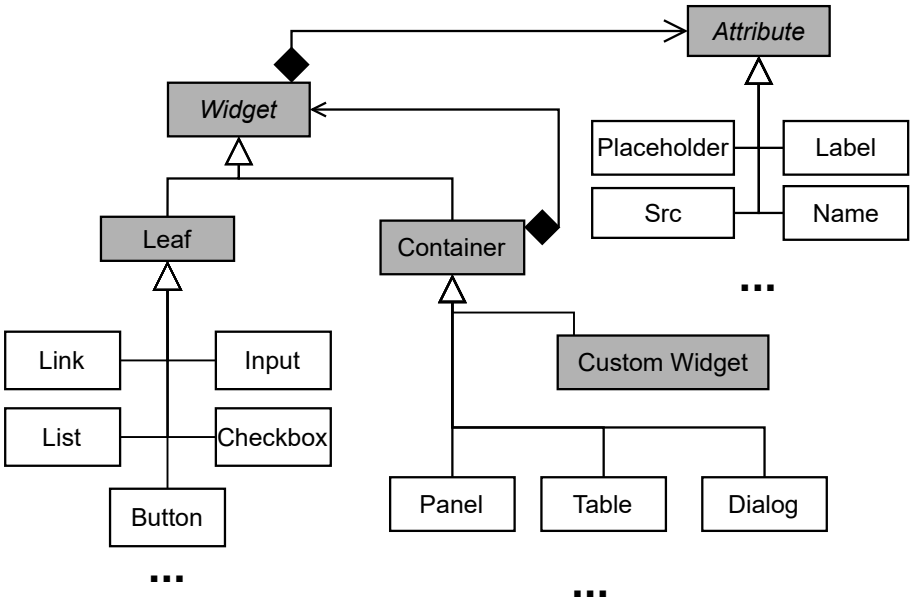


Figure 3.7: Excerpt of the widget package

This package can be extended with other widgets to fit the needs of a specific migration. Extending this package is an option when developers have developed specific widgets for their company and need to migrate them. The widget package includes the already known widgets, whereas the `Custom widget` represents the unknown widgets.

[4] https://www.w3schools.com/html/default.asp

[5] The complete Pivot meta-model is presented at https://badetitou.fr/projects/Casino/#full-widgets-meta-model

3.3.1.3 Layout package

To represent the layout of a graphical user interface, we designed a dedicated package. It allows one to represent the visual disposition of the graphical components of the user interface. Our layout meta-model allows one to represent any hierarchical layout, which is the most common one (see Section 2.1.2).

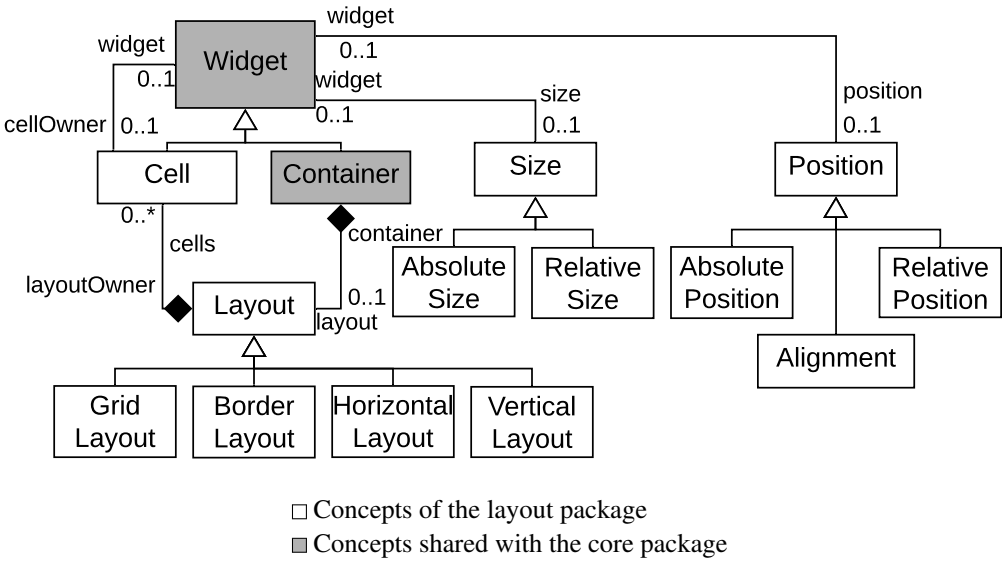


Figure 3.8: Layout package

Figure 3.8 represents our layout package. The entities `Widget` and `Container` are part of the core package presented in Section 3.3.1.1. The layout package adds four main entities to the core package.

`Size` describes the height and width of a widget. The size of a widget can be absolute or relative. The `AbsoluteSize` is expressed in pixels. The `RelativeSize` is expressed as a percentage of its container.

`Position` describes the position of a widget in the user interface. It can be absolute, relative, or defined by alignment properties. The `AbsolutePosition` represents the coordinates of a widget in the user interface. The `RelativePosition` represents the coordinates of a widget in its container. The `Alignment` defines how to position a widget inside its container. It can be in the *top*, *bottom*, *right*, *left* or *center* of its container or a combination of them.

`Layout` represents rules to position the children of one container. Any `Container` of the core package can have one layout. A `Layout` can

be refined as a `Grid Layout`; a `Border Layout`; a `Horizontal Layout` and a `Vertical Layout`. We currently support these layouts because they are most frequently used in our context. However, one can extend the layout package to support other hierarchical layout managers.

`Cell` A `Layout` can contain multiple `Cells`. Then, each `Cell` contains one widget. Thus, the layout arranges the widgets using `Cells`. It allows one to fine control the final GUI layout.

Note that some `Containers` do not have a `Layout`. For instance, a <select> in HTML has multiple <option>, thus, it is considered as a `Container` but does not have `Layout`.

3.3.1.4 Custom widget

One of the challenges when considering GUI framework migration is the ability to handle widgets that might not be present in the pivot meta-model. Indeed, widgets in a source framework might be absent of the target framework [Shah and Tilevich, 2011, Gerdes Jr, 2009, Sánchez Ramón et al., 2014, Sánchez Ramón et al., 2016]. For example, AWT is an old GUI framework, and some of its widgets do not have a counterpart in Angular. It is the case of the MenuBar Java class[6] that corresponds to a toolbar positioned at the top of a window. It is a common widget in AWT but must be recreated in Angular. Note that in the case of migration between standard web applications, the problem is less important because most of the widgets are also standard (*i.e.,* , , <input>, *etc.*).

To tackle such a problem, we use the concept of `Custom Widget` [Sánchez Ramón et al., 2014, Sánchez Ramón et al., 2016]. When an unknown widget type is encountered during the extraction step, the extractor creates a `Custom Widget`. Then, it extracts the DOM of the `Custom Widget` as if it was a container (*i.e.*, in HTML, or Container in Java Swing). During the generation, `Custom Widgets` are generated as generic containers with a comment in the generated code to warn developers and give them additional information: name of the source widget, attributes (if identified), possible children widgets (if identified), location in the source code. With this information, developers can manually add a new widget to the Pivot meta-model and update the known widgets mapping (mapping source to pivot).

We designed our approach and the meta-model to be specializable. Thus, to avoid migrating unknown widgets as a generic container using the `Custom Widget`, one can create specific widgets and use them with our approach. The added widgets will then be migrated. It also enables our approach to be iterative:

[6] MenuBar AWT javadoc: https://docs.oracle.com/javase/8/docs/api/java/awt/MenuBar.html

one performs the migration, our tool identifies `Custom Widgets`, then the developers extend our meta-model and iterates. Note that such a widget should also be created (programmed) in the target framework to improve migration results. If the developers felt the need to create them in the source framework, there is a good chance that the same need applies to the target framework.

We presented the three Visual code packages that compose our Pivot meta-model representation (see Figure 3.4). It allows us to present the DOM, the different widgets, and their layout. We also detailed the `Custom Widget` functioning that deals with unknown widgets. In the following, we present the steps to build a model instance of this meta-model from an application and how it can be used to generate the target application GUI.

3.3.2 Visual code extraction

The Visual code migration consists of extracting the widgets, their attributes, and their layout and linking them together. Custom widgets defined by the developers and used in the applications should also be managed. We divided the Visual code extraction into several steps. Some steps are framework dependent, others are application dependent. Framework dependent steps must be performed to support a new source GUI framework. Application dependent steps must be performed for each migration project.

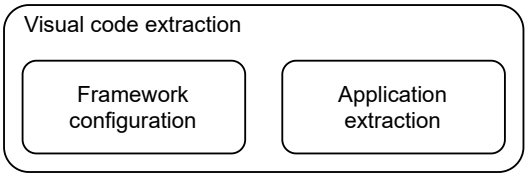


Figure 3.9: Visual code extraction steps

Figure 3.9 presents the two steps to extract the Visual code. In the following, we describe the first step applied (once) for each new GUI framework in Section 3.3.2.1. Then we describe the second step applied for each application of an already known framework in Section 3.3.2.2.

3.3.2.1 Visual code extraction — framework configuration

To extract an application's GUI, one must first configure our approach to support the GUI framework it uses. The result of this step will then be used for every migration project using the same source GUI framework. It consists of analyzing the source GUI framework and defining mapping and rules.

Table 3.1: Sub-steps to support the extraction of a new GUI framework

source	Map source framework to Pivot meta-model	Indentify containment rules	Indentify custom widgets rules	Indentify root widgets rules
Markup language	Map tags	Use DOM	Unknown tag	Configuration file
Programming language	Map source classes and factories	Use method invocation	Unknown subclasses	Specific set of widgets

To support a new source framework (see Table 3.1) we define the following sub-steps:

1. **Map source framework to Pivot meta-model**, where we define dictionaries mapping known widgets, attributes, layouts of the framework to their counterpart in our Pivot meta-model. For programming languages, we map the widget class and the factory methods to their pivot counterparts. For example, in Swing, JButton maps to our pivot `Button` widget of the widget package (see Section 3.3.1.2 and concrete example Section 3.2.1), and .setLabel() is mapped to our pivot `Label` attribute. We also map factory methods to their widget counterpart. For instance, FactoryButton.create(Object obj) is a factory that create a widget button depending on the obj parameter. We map the method create of FactoryButton class to our pivot `Button` widget. For markup languages, we map a tag to a widget concept. For example, in HTML, <button> maps to our pivot `Button` widget, and the label attribute maps to our pivot `Label` attribute.

2. Then, we need to **identify containment rules** in the source code. Containment links together widgets/layout and their children attributes/widgets. In the case of programming languages, such containment links may come from specific method calls as add(...) or setWidget(...) on a container widget. For attributes, identifying the containments might use the setter methods as for the previous sub-step. For instance, in myButton.setLabel("aLabel"), setLabel is mapped to the pivot `Label` attribute and its receiver (myButton) is the attribute owner. For markup languages, the containment is already defined in the DOM.

3. **Identify custom widgets rules** specifies how to identify application-specific widgets that are not part of the *source to pivot* dictionary map defined in the first step. It means identifying that something is a widget even though we do not know this widget. For programming languages, it generally corresponds to unknown subclasses of a generic widget class. For example, the rule for GWT is to look for all subclasses of the Widget class; in Angular, one looks for all component.ts files. For markup languages, we look for unknown tags.

Such custom widgets can typically not be translated automatically but need to be identified. Thus, the generator flags them in the generated GUI for developers to take actions (either migrate them manually or update the *source to pivot* map, see Section 3.3.1.4).

Note that we only care about custom widgets and not custom attributes or layout. We consider it impossible to define custom attributes (that would apply to already known widgets) or custom layouts. Such new attributes/layouts can only come as part of new custom widgets.

4. Finally, **identify root widgets rules** specifies how the root widgets will be recognized. Root widgets are the root of the DOM defining the application's GUI. They correspond to windows in a desktop application or pages in a web application. For markup languages, roots are defined in a configuration file, whereas in programming languages, they are identifiable as a specific set of widgets (*e.g.*, JWindow in Java Swing). Later, we will see that they are essential to build the hierarchical representation of a GUI.

We detail the implementation of these sub-steps in Section 4.1.1.

3.3.2.2 Visual code extraction — application extraction

Once our approach is "configured" to support a new GUI framework, one can extract the Visual code of an application using this GUI framework. Again, sub-steps must be considered for the migration of each application. Contrary to the ones described in the previous section, these sub-steps should be adapted to every migration project, even if the applications use the same GUI framework. These sub-steps are inspired by the ones described in the literature (Section 2.2.4).

Table 3.2: Sub-steps to extract Visual code using a known framework

Source	Identify Custom Widget	Create widgets instances	Detect composition	Perform layout additional sub-step
Markup language	Apply Custom widgets identification rules	Browse markup file and create widgets when encounter recognized tags	Children in the GUI are children in the DOM	—
Programming language		Look for source class instantiation or call to factories	Look for call to pre-determined methods	Use symbolic execution to resolve precise position of widget

In the following, we present the four sub-steps for the extraction of the visual code part of the GUI migration. These sub-steps actually perform the extraction of application GUI based on the rules defined for the GUI framework. We also detail how to adapt each sub-step for GUI based on programming languages and markup languages. Table 3.2 summarizes the sub-steps and how we adapt them.

1. First, **identify custom widgets** is based on the rules for the framework (*custom widget rules*). This sub-step actually performs the identification of the custom widget in the application to migrate. For both markup and programming languages, it corresponds to applying the custom identification rules defined in the third sub-step of the preceding section. The new widgets are added, on the fly, to the framework *source to pivot* dictionary. Each *instance* of unknown widgets is mapped to a different instance of `Custom Widget`. No effort is made to group various instances of the same unknown widget together.

2. Second, in the **create widget instances** sub-step, the *source to pivot* dictionary for the framework defined in the previous section is used to identify all instances of known widgets, attributes, and layouts. We create for each instance its equivalent in our GUI pivot model. For markup languages, we visit the markup source file and create widgets corresponding to recognized tags. For programming languages, we look for widget class instantiations. They can occur by calling the widget constructor (*e.g.*, in Java: `new`) or through a factory defined by developers or the source framework. This strategy is similar to the one proposed in the litterature (see Section 2.2.4).

3. Third, **detect composition** is based on *identified containment rules* of the previous section sub-step. This sub-step links each instance of widgets, attributes, and layouts with its parent widget. As a result, this sub-step extracts the DOM of the GUI. In markup languages, the composition is already defined in the DOM, so children in the source DOM are children in the pivot DOM. For programming languages, we look for call to methods defined in the *containment rules* (*i.e.*, `add(...)`, `setWidget(...)`). This sub-step results in a DOM that includes the widgets, their attributes, and also the layouts.

4. Finally, **performing a layout additional** sub-step is often necessary to improve the computation of widgets layout (typically with grid layouts). For example, widgets could be positioned one relative to the other, or some computation might be required to get the row and column values in a grid layout. In markup languages, information such as the row and column values are already provided by the DOM. Thus, there is no need for this sub-step for markup languages. However, this sub-step is necessary for programming languages and might require symbolic execution to resolve widgets' position in the GUI. Note that using symbolic execution to better extract application GUI was proposed in the literature Section 2.2.4.

We detail the implementation of these sub-steps in Section 4.1.2.

3.3.3 Visual code generation

There is no study nor detailed explanation in the literature on how to export Visual code into the target language (see Section 2.2.4). The basic approach consists of visiting the DOM of the pivot model and generating appropriate code. However, the generated code may be split into different files, or one pivot entity may produce several target entities, or several pivot entities may be grouped in only one target entity.

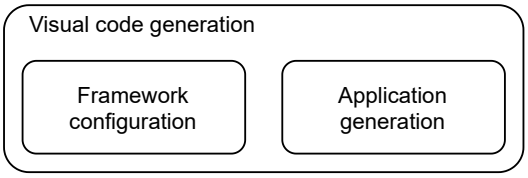


Figure 3.10: Visual code generation sub-steps

As for the extraction, the generation is split into the configuration for new GUI frameworks and new applications. Figure 3.10 presents the two steps of the Visual code generation. The steps that configure our approach for a new target GUI framework are detailed in Section 3.3.3.1. Then we discuss actual generation of an application using a known target framework in Section 3.3.3.2.

3.3.3.1 Visual code generation — framework configuration

As for the Visual code extraction (see Section 3.3.2.1), the first step for the generation is to configure our approach to support the target GUI framework. This configuration will then be reused for every application generation using the same target GUI framework. It consists of analyzing the target GUI framework and defining mapping and rules.

Table 3.3: Sub-steps to support the generation using a new GUI framework

Source	**Identify target framework environment**	**Map Pivot meta-model to target framework**
Markup Language	Written in one or multiple files	Widget concept to tag
Programming Language	Defined in one method or multiple methods according to the target framework	Widget concept to widget instantiation method

Two sub-steps are needed to support a new framework (see Table 3.3): *identify target framework environment* and *map Pivot meta-model to target framework.*

1. The first sub-step, **identify target framework environment**, defines where the code will be generated to be supported by the target GUI framework. As a result, the generator is configured to generate the target code in compliance with the target GUI framework requirements. For instance, for markup languages, the GUI can be fully defined in a file (*.html*) or multiple files (Angular components). In the case of programming languages, some frameworks force the user to define the GUI in a specific method, or the GUI can be defined at any place in the code (Java Swing).

2. In the second sub-step, **map Pivot meta-model to target framework**, we define a mapping between widgets, attributes, and layouts to their target framework counterpart. For markup languages, it corresponds to the target tag, and for programming languages, the way to instantiate the widget or set the attribute (*i.e.*, calling the constructor or a factory).

We detail the implementation of these substeps in Section 4.2.1.

3.3.3.2 Visual code generation — application generation

Once the support of the target framework is configured, it is possible to generate an application. The generation uses the mapping between the pivot meta-model and the target framework and the identified target framework environment to supervise the target application code generation.

Table 3.4: Sub-steps to generate the Visual code using a known framework

Source	**Identify target application environment**	**Generate Code**
Markup Language	Configuration information (URL for web application data access)	Visit the pivot model DOM and generate for each widget/attribute/layout/ its tag counterpart
Programming Language		Generate the GUI code using setter, DOM builder methods, and constructor

Two sub-steps are needed to support the generation of a new application (see Table 3.4): *identify target application environment* and *generate code*.

1. **Identify target application environment** is identical for programming and markup languages. It consists of configuring the generator with the target application environment settings. For example, by setting up the back-end URL endpoints, the generator can generate calls to back-end services and set the path to the image source for image widgets.

2. The second sub-step, **generate code**, defines how the code is exported. Whereas configuring how the code generator visits the GUI model to generate the code is defined at the "support new framework" level, one must perform the code generation for each migration project. For target markup languages, it consists of visiting the pivot model DOM. For each widget, the generator creates its target language counterpart with its attributes. In the case of generating programming languages, it calls methods that instantiate the widgets (*e.g.*, call to the constructor, call to a factory, *etc.*) and the methods used to build the DOM (*i.e.*, add(), setWidget()) to generate the target GUI.

We detail the implementation of these sub-steps in Section 4.2.2.

We detailed the sub-steps to extract and to generate the Visual code. Using these sub-steps, one can migrate the visual aspect of an application. However, end-users will not be able to interact with the UI. To enable interaction, one must before migrate the Behavioral code.

3.4 Behavioral code migration approach

Based on the Behavioral code description and the literature, we designed a Behavioral code migration approach. First, Section 3.4.1 describes our Behavioral code package linked to the core package (see Figure 3.4). Then, Section 3.4.2 presents the extraction steps. Finally, Section 3.4.3 presents the generation steps.

3.4.1 Behavioral code package

To represent the Behavioral code, we designed a behavioral package. It is based on an AST meta-model to represent as closely as possible the executed code. It comes as an extension of the FAST meta-model[7] a generic AST meta-model. The package is divided into two parts: the `Events` raised by user interaction, and the `Manipulation code`.

To integrate all the behavioral concepts inside the generic AST, we defined Manipulation code entities as `ASTExpressions`. Thus, we can attach our behavioral model to any AST model using the `ASTExpression` concept.

In the following, we present the concepts of our behavioral package illustrated in Figure 3.11. `Events` are represented at the right of the figure; and `Manipulation code` at the center and left of the figure.

`Event` corresponds to the events that will be raised when the end-user interacts with the UI. It can be refined as `Click`, `Change`, `Error`, `Submit`, and

[7]FAST (generic AST): https://github.com/moosetechnology/FAST/

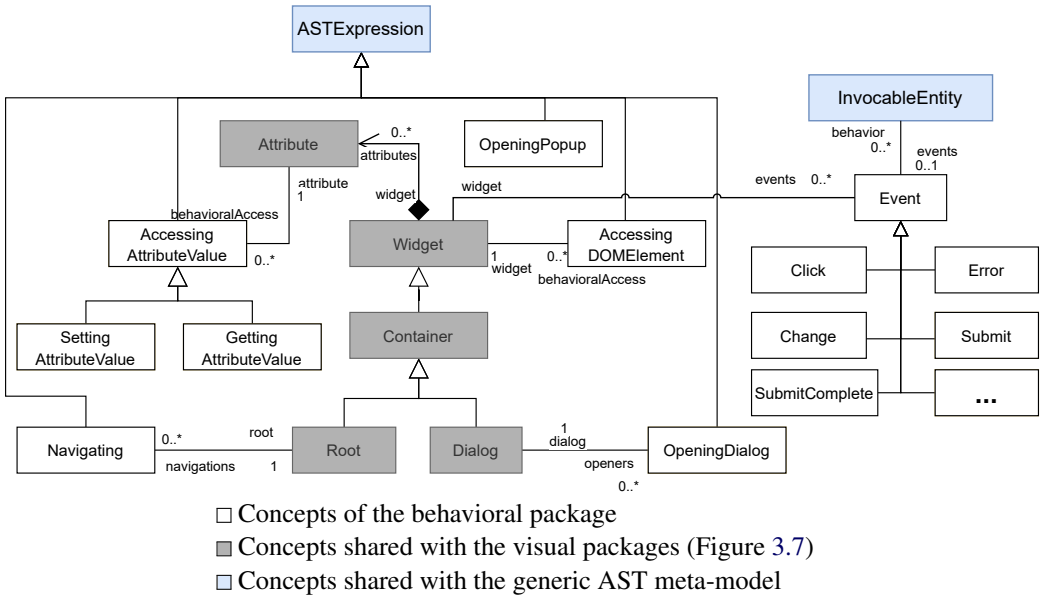


Figure 3.11: Behavioral package

`SubmitComplete` or any other event (see Section 3.1.2.2). An `Event` is linked to the AST concept `InvocableEntity` which represents an element that can be invoked, *e.g.*, a method or a lambda expression. An `Event` is also linked to a `Widget` on which it is attached.

`AccessingDOMElement` represents the reference to any widget of the DOM. For example, in myInput.value, there is the access to the Java variable myInput. Assuming this variable contains an `Input` widget, there is an access to the `Input` widget from the Behavioral code. Note that, a widget can have multiple references.

`AccessingAttributeValue` represents an access to a widget attribute, and so is linked to the `Attribute` concept of the core package. It can be refined as a `GettingAttributeValue` or a `SettingAttributeValue`.

`Navigating` corresponds to the Manipulation code to navigate from one page of the application to another. This concept is linked to the `Root` concept of the core package. This piece of Manipulation code is the most represented one in the literature.

`OpeningPopup` corresponds to the code executed to open a Popup.

`OpeningDialog` corresponds to code executed to open a dialog. The `Dialog` concept is already defined in the widgets package (see Section 3.3.1.2), and several `OpeningDialog` can be associated to the same `Dialog`.

We presented the Behavioral code package and its integration with the Visual code packages and a generic AST meta-model. Using this representation, we now present how to perform the Behavioral code migration.

3.4.2 Behavioral code extraction

Because the Behavioral code package is linked to the Visual code packages (see Section 3.4.1), the first step to extract the behavioral code is to extract the Visual code model (Section 3.3).

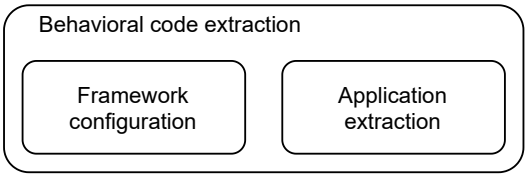


Figure 3.12: Behavioral code extraction sub-steps

Then, it is possible to extract the Behavioral code model.

As for Visual code extraction, we split the steps to support a new framework and a new application. Steps to support a new framework are presented in Section 3.4.2.1 and to actually extract Behavioral code from an application in Section 3.4.2.2.

3.4.2.1 Behavioral code extraction — framework configuration

Two sub-steps must be performed to configure the Behavioral code extraction for a new framework. The first one, *Identify event handler types*, is performed to support event extraction, and the second one, *Define Manipulation code patterns*, is performed to support Manipulation code extraction.

We now detail the event extraction step.

Identify event handler types: We identify all the possible event handlers that exist in the source application framework. Example of event handler types are: *click*, *change*, *hover*, ...For markup languages, the existing handler types are defined in the documentation[8] of the GUI framework. For programming languages, a handler is a class or a lambda responsible for executing the Behavioral code when an event occurs.

In addition to the event extraction configuration, we configured the extraction of Manipulation code. Note that, since Manipulation code is expressed only in programming languages, we only report configuration for programming languages.

[8]For plain HTML: https://developer.mozilla.org/en-US/docs/Web/Events#event_listingl

Define Manipulation code patterns: We define patterns to identify Manipulation code inside the source code AST model (see Section 3.4.1 and Figure 3.5). The different kinds of Manipulation code can be detected by one or multiple patterns. Since Manipulation code depends on the GUI framework, one must redefine patterns for each framework, but patterns are common to all applications that use the same framework.

Table 3.5: Sub-steps to configure new framework extraction for Behavioral code

Source	Identify event handler types	Define Manipulation code patterns
Markup Language	Tag attributes to handler concept	N/A
Programming Language	Source class	Define patterns from manual migration examples

Table 3.5 summarizes the sub-steps needed to support a new framework. We present implementation examples of these sub-steps in Section 4.3.1.

3.4.2.2 Behavioral code extraction — application extraction

Once we configured our approach for a new GUI framework, we can extract the Behavioral code of an application using this GUI framework. Again, we split this step into sub-steps to extract events and sub-steps to extract Manipulation code.

Two sub-steps are required to extract Behavioral code events: *Detect event handler instances*, and *Attach event handlers to widgets*.

Detect event handler instances: We determine where, in the source code model, the event handlers are created (*i.e.*, instantiations of the event handler types). For markup languages, they are represented by specific kind of attributes. For example, in <button onclick="myFunction()"> the attribute onclick is a special attribute that creates handler instance. For programming languages, it corresponds to class instantiations or lambda definitions.

Attach event handlers to widgets: We link the handler instances from the previous sub-step to their widget owners. This sub-step is similar to the *Detect composition* sub-step for Visual code extraction (see Section 3.3.2.2). From this sub-step, we know the interactions allowed by the application for each widget. For markup languages, the handler is attached to the widget that holds the handler attribute. Again, in <button onclick="myFunction()">,

the onclick handler is attached to the button widget. For programming languages, the handler is attached to a widget using a specific method invocation. For example, button.addActionListener(...) in Java Swing or button.setOnAction(...) for JavaFX. In both cases, the handlers are attached to the button widget.

To finalize the Behavioral code extraction, two more sub-steps are necessary. These sub-steps aims to extract Manipulation code: *Apply Manipulation code patterns*, and *Manipulation code model transformation*.

Apply Manipulation code patterns: Based on the extracted event handlers detected in previous sub-steps, our tool analyses the source code model (see Figure 3.5) to identify the methods executed when an event is fired. Then, we use a pattern matcher on the identified methods ASTs with the patterns defined in the previous section (see Section 3.4.2.2). It provides the location of the Manipulation code in the source code model.

Manipulation code model transformation: Then, we apply model transformations on Manipulation code location. It consists of creating the behavioral entity and its associations associated with the pattern (*e.g.* `OpeningPopup` or `Navigating`) and replacing the old AST expressions with the new GUI behavioral entities.

Table 3.6: Sub-steps to extract application Behavioral code

Source	Detect event handler instances	Attach event handlers to widgets	Apply Manipulation code patterns	Manipulation code Model Transformation
Markup Language	Usage of the attribute in the DOM	The widget corresponding to the attribute hosting tag	—	—
Programming Language	Class instantiation	Method invocation	Use a patern matcher to determine Manipulation code position	Perform the model transformation

Table 3.6 summarizes the sub-steps needed to extract Behavioral code of an application. We present implementation examples of these sub-steps in Section 4.3.2.

Once the Behavioral code is extracted in a model, it is possible to generate in the target framework.

3.4.3 Behavioral code generation

Again, we split the generation into steps to support new frameworks (Section 3.4.3.1) and steps to support new applications (Section 3.4.3.2).

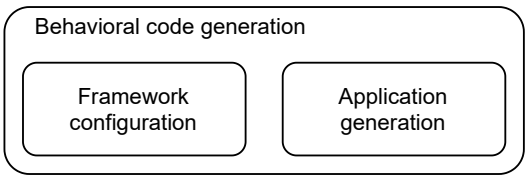


Figure 3.13: Behavioral code generation sub-steps

3.4.3.1 Behavioral code generation — framework configuration

To support the generation of Behavioral code, we first defined two sub-steps configuring our approach to support a new target framework. The first one deals with events generation and the second one with Manipulation code generation. Whereas their implementations are different, the two sub-steps consist of creating the mapping between our Pivot meta-model and the target framework. These sub-steps are similar to the second sub-step performed when supporting a new framework for Visual code generation (see Section 3.3.3.1), *i.e.* they map concepts from frameworks to our Pivot meta-model.

We now detail the *Map pivot events to target framework* sub-step that configures our approach to generate events using a target framework. This sub-step is performed in the same way for target frameworks based on markup languages or programming languages.

Map pivot events to target framework: We map the event handlers represented in our Pivot meta-model to the available event handlers in the target framework. This step is done only once per target framework and highlights the handlers existing in the source framework and our Pivot meta-model but not in the target framework. For such handlers, a specific approach should be designed. One solution is to redevelop the handler in the target framework. This solution is similar to migrating custom widgets (see Section 3.3.1.4).

In addition to the pivot events to target framework mapping, one has to perform a similar mapping for Manipulation code.

Map Manipulation code to target framework: The sub-step consists of mapping our pivot Manipulation code to its target framework counterpart. To do so, we propose first to perform part of the migration manually to discover the target code that corresponds to the source Manipulation code. Then, it is possible to fill the mapping with Manipulation code and its textual target counterpart.

Table 3.7 summarizes the sub-steps needed to configure the generation of Behavioral code for a new framework. We present implementation examples of these sub-steps in Section 4.4.1.

Table 3.7: Sub-steps toi configure new framework generation for Behavioral code

Source	Map pivot events to target framework	Map Manipulation code to target framework
Markup Language	Map each Pivot meta-model handler to its target framework counterpart	—
Programming Language		Map Manipulations code to their textual target framework representation

3.4.3.2 Behavioral code generation — application generation

Once we configure our approach for the target framework, we can generate the target Behavioral code. Again, the generation is divided between the events and the Manipulation code generation.

Based on the mapping between pivot events and the target framework built in the previous section, we perform the *Generate event handlers* generating event handlers in the target application.

Generate event handlers: We generate the event handler in the target code. To do so, we extend the *generate code* Visual code generation step (see Section 3.3.3.2). Indeed, it consists of adding into the visual generated code instructions to set the event handlers.

To finalize the generation of the Behavioral code, the last sub-step consists of the generation of the target code, including Manipulation code.

Generate target AST: Finally, we generate the target code from the AST attached to each extracted event handler. To do so, we generate the code of the method called by the event handler, add the method dependencies (*e.g.*, import statements, variable initializations, *etc.*), and add potential comments in the generated code for missing migration rules.

Table 3.8: Sub-steps to generate application Behavioral code

Source	Generate event handlers	Generate target AST
Markup Language	Add in the generated markup tag the handlers attribute	—
Programming Language	Generate the event handlers setter attached to its widget	Visit AST linked to handlers and generate code associated to each node

Table 3.8 summarizes the sub-steps needed to generate the Behavioral code of an application. We present implementation examples of these sub-steps in Section 4.4.2.

The generation of the Behavioral code completes the migration of the Visual code. After the migration of the events and Manipulation code, end-users get access to an application developed using the target framework that includes the GUI of the source application.

3.5 Conclusion

In Chapter 2 we presented the existing GUI representations and solutions to migrate GUIs using one GUI framework to another GUI framework. Whereas existing solutions offer good results, they also fail to tackle two major challenges: having a complete representation of the GUI and enabling the adaptation of their work to other contexts.

To tackle these challenges, we, first, proposed in this chapter a definition of the GUI. It includes a separation of the GUI inspired from the literature into the Visual code that deals with the visual element and their disposition on a page; the Behavioral code that deals with the end-user interaction with the GUI; and the Business code that deals with application manipulated data. Together, Visual code, Behavioral code, and Business code aim to represent the GUI of an application. Thus, they tackle the first challenge.

From this definition of the GUI, we defined a generic approach to migrate application GUIs. Following the literature's classic horseshoe process, the approach is subdivided into steps and sub-steps to migrate a GUI. To ease this approach adaptation to different contexts, we detailed each sub-step with possible adaptation for frameworks based on programming languages or markup languages. Figure 3.14 illustrated our approach and the main steps for extraction and generation of Visual code and Behavioral code. Using this approach and its possible adaptation, we tackle the second challenge.

Our definition and approach allow one to migrate the GUI of an application. We presented solutions to adapt our work to different frameworks. To better understand how to implement our approach in a tool for different GUI migration projects, we present several implementations of our approach in the next chapter.

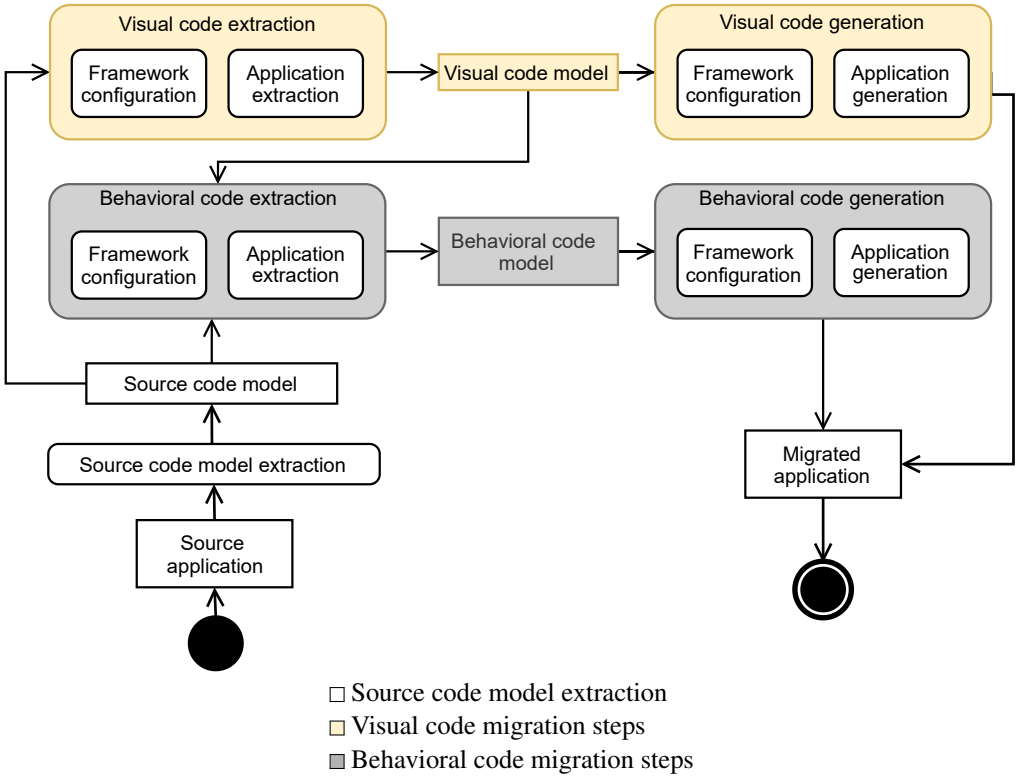


Figure 3.14: Our GUI detailed migration process

CHAPTER 4

Implementation

Contents

4.1 Visual code extraction 55
4.2 Visual code generation 61
4.3 Behavioral code extraction 63
4.4 Behavioral code generation 68
4.5 Conclusion 73

This chapter aims to provide examples of our GUI migration approach implementation for different contexts. We implemented Visual code extractors and generators for GUI frameworks that use programming and markup languages and a Behavioral code extractor and generator for the GWT GUI framework based on the Java programming language.

We present our implementations for the Visual code and Behavioral code migration separately. Section 4.1 details the extractors' implementation of the Visual code. Section 4.2 focuses on the Visual code generators' implementations. Section 4.3 details the Behavioral code extractor implementation. Finally, Section 4.4 details the Behavioral code generator implementation.

4.1 Visual code extraction

We implemented three extractors for three different frameworks: The first one, BLCore/GWT, enables designing web GUI in the Java programming language. The BLCore framework is the one used by our industrial partner and is an extension of the GWT GUI framework. The second one, GXT[1], enables designing web GUI using the XML markup language. GXT and GWT applications contain configuration files written in XML to describe some application parameters, such as the main pages, the URL and the port used when deploying the application, and the URL of the back-end. The last one, Spec [Fabry and Ducasse, 2017], enables designing

[1]GXT, https://www.sencha.com/products/gxt/, is also an extension of GWT

desktop GUI in the Pharo[2] programming language. Each extractor builds a pivot model from an application.

In this section, we detail the steps of the visual code extraction (see Figure 3.14). We discuss separately the sub-steps to support a new framework (see Section 3.3.2.1) and the sub-steps to actually extract Visual code of an application (see Section 3.3.2.2) with concrete examples from the extractors.

4.1.1 Visual code extraction — framework configuration

To support a new framework (see Section 3.3.2.1), one needs to:

- Map the source framework to the Pivot meta-model,
- Identify the containment rules,
- Identify the custom widgets rules, and
- Identify the root widgets rules.

Map source framework to pivot model. The basic approach to define a mapping between source and pivot meta-model is to create a dictionary. For GXT extractor, we map an XML tag or attribute (source meta-model) to its corresponding pivot widget, attribute, or layout. For example line 2 of Listing 4.1, the container:VerticalLayoutContainer source tag maps to the pivot widget Panel containing a Vertical pivot layout. The pivot Panel concept is defined in our widget package (see Section 3.7) and the Vertical pivot layout in our layout package (see Section 3.3.1.3).

```
1 <gxt:Window ui:field="window">
2   <container:VerticalLayoutContainer>
3     <form:FieldLabel text="{i18n.user}">
4       <form:widget>
5         <form:TextField/>
6       </form:widget>
7     </form:FieldLabel>
8   </container:VerticalLayoutContainer>
9   <gxt:button>
10    <button:TextButton text="{i18n.login}"/>
11  </gxt:button>
12 </gxt:Window>
```

Listing 4.1: Snippet of an GXT login view in XML

[2] https://pharo.org/

For the frameworks based on programming languages (GWT and Spec), we map a source class or method to a pivot widget, attribute, or layout. We need to consider also methods because some widgets may be constructed through factories. Also, an attribute of a source widget might be assigned with a setter method; in this case, the source method (setter) maps to a pivot attribute. We already gave mapping examples in Section 3.3.2.1. We do not consider getters for the GUI extraction because they do not tell us anything about the value to assign to the attributes. Therefore they do not point us to anything that could help generate code.

There are other possible mappings, for example a GWT source widget DynamicFieldSetPanel maps to a pivot widget FieldSet and its boolean attribute dynamicFieldSet. The presence of an attribute may also be conditioned to the instantiation of a widget. For example, instantiating a Button widget with a string parameter (new Button("OK")), will set its text attribute. Another case is that of a source attribute mapping to two pivot attributes. For example the width source attribute maps to either the absoluteWidth or relativeWidth pivot attributes depending on whether the value assigned to it is in pixel (*e.g.,* "250px") or in percentage (*e.g.,* "50%"). Identifying such differences requires symbolic execution and cannot always be achieved (see Section 4.1.2).

Identify containment rules. The containment link rules identification might be straightforward. For example, in GXT, we use the XML file's DOM describing the interface since it is already structured as a containment tree.

The links between a widget and its attributes are easily set for frameworks based on programming languages when the attributes are identified. Indeed, when an attribute is defined in a widget constructor, this attribute is represented as contained by the widget. For instance, in new Button("OK"), the text attribute is linked to the Button widget. When the attribute is defined with a setter method, the attribute owner is the receiver of the method invocation. For instance, in input.setName("a Name"), the name attribute is linked to the Input widget.

The containment links between widgets and sub-widgets, or widgets and layouts, are identified through the use of a small set of specific methods: ownerWidget.add(<*widget*>) for Spec and GWT, and also ownerWidget.setWidget(..., ..., <*widget*>) for GWT. For these methods, we specify that the parent widget is the receiver and the child widget is the argument.

Finally, in some GUI frameworks, widgets and layout are mixed, it is the case for GWT with HorizontalPanel class that mixes the Panel widget and the Horizontal layout (see Section 2.1.2).

Identify custom widget rules. In programming language based frameworks, we typically look for new classes inheriting from the most abstract widget in the source language: Widget class for GWT, and ComposablePresenter for Spec.

In GXT, custom widgets are either unknown tags in a GUI description file (named xxx.ui.xml) or a GUI description file not listed as root in the configura-

tion file. For example, in the snippet of Listing 4.1, we expect to know all tags within <gxt:Window>. Any unknown tag, *i.e.* not present in our widget to pivot concept mapping defined in the *Map source framework to pivot model* sub-step, inside <gxt:Window> is then considered as a custom widget.

Identify root widget rules. In GWT and GXT, we browse the XML configuration file describing the application where all root widgets are listed. Note that this is the same file in both cases as GXT is just an extension of GWT.

In Spec, the notion of root widget is fuzzier because it relies on the idea that any widget can be opened as a window or included in another widget. Therefore, in this case, we rely on the user to tell us what the root widgets are for each application.

4.1.2 Visual code extraction — application extraction

As presented in Section 3.3.2, to migrate a new application in a known framework, one needs to:

- Identify the custom widget types,
- Create all widget instances,
- Detect the composition of widgets, and
- Perform an optional additional sub-step for layout.

Identify custom widget. We apply the identification rule defined for the given framework. For example, as described above, in GWT we look for all new class descendants of Widget.

Create widget instances. In GXT, identifying widgets instances, attributes, and layout is achieved by browsing the XML file from top to bottom, creating pivot counterpart as we encounter tags (these tags were identified in the Mapping Definition sub-step in the previous section). The composition of the widgets and their attributes is extracted from the source XML file's DOM. The same goes for layouts.

Identifying widgets, attributes, and layout instances for applications using frameworks based on a programming language is more complex. In the following, we present the example for GWT based on Java, but the very same approach is applicable to Spec based on Pharo.

If a source widget is identified by a class, we just look for instantiations (new) of this class. If a source widget is identified by a method (*e.g.*, in a factory), we look for calls to this method. In both cases, we need to keep a reference to the instance created for later analysis. Typically the instance is assigned to a variable, and we retain this variable. For example, in Listing 4.2, line 4, the LinkLabel instance is

```
1  class SPBusiness1 extends AbstractBusinessPage {
2    @Override
3    public void buildPageUi(Object object) {
4      LinkLabel lblPg = new LinkLabel("Next");
5      lblPg.setEnabled(methodCall());
6      content.add(new Label("<Business content>"));
7      content.add(lblPg);
8      super.setBuild(true);
9    }
10 }
```

Listing 4.2: User interface creation in Java GWT

assigned to the variable lblPg. Since we are working with a source code model and not the source code as a textual artifact, we have the variables as entities in the source model (see Section 3.2.3). Thus, we can easily find every place where a widget is accessed. Finally, in Listing 4.2, there are two widget instantiations: LinkLabel, line 4; and Label (another known widget), line 6.

When looking for attributes in the source model, we search for known setter messages sent to variables containing widget instances. Again these setter messages were identified for the framework as indicators of attributes. In Listing 4.2, on line 5, the lblPg variable receives the setEnabled message that maps to the disable pivot attribute. Note that, again, the argument of setEnabled must be interpreted to give the correct value to the pivot attribute. In the example, we have to execute the method methodCall to resolve the boolean value. It is one of the most complex and least reliable computations we perform, as will be seen in the results of experiments given in Section 5.1.3. Nevertheless, we still achieved a worst-case of 67% of attributes correctly detected.

Detect composition. For the DOM building, there are two examples of calls to the add(*<widget>*) method in Listing 4.2, on line 6 and 7. In each case, we already identified the variable to which these messages are sent (content) and the children widgets that are passed as arguments.

Note that the line 8 of Listing 4.2 is not used as it does not involve any known variable or method. It actually does not impact the interface built.

In Spec, the process is similar except that widget creation and containment links are defined at different places. Still, they also rely on the use of variables containing the widgets created and used later to establish the containment links. The work is more straightforward as containment links are separated in a well-defined method, making it a bit similar to a declarative specification.

Listing 4.3 presents a defaultSpec method in Pharo. It is the method that developers have to extends to define the DOM of a GUI with the Spec GUI framework.

```
1 defaultSpec
2   <spec: #default>
3   ^ SpecLayout composed
4       newColumn: [ :col |
5         col newRow: [ :row |
6           row add: #buttonNormal.
7           row add: #buttonDisabled.
8         ]
9       ];
10      yourself.
```

Listing 4.3: Building layout and DOM in Pharo Spec

Line 4, a vertical panel is created (newColumn:) and line 5 a horizontal panel is created (newRow:). Line 6 and 7, two widgets contained in variables, buttonNormal and buttonDisabled, are added to the horizontal panel. Using the method newColumn:, newRow:, and add:, the DOM building of Spec GUI framework is similar to the one used by markup based GUI framework such as GXT.

Note that, in our Pivot meta-model, children widgets are rarely added directly in their parent widget. Because our DOM also contains the layout, widgets are added to `cells` that are put in `layouts`, themselves children of the parent `widget`.

Perfom additional layout computation. To accurately represent the layout, one needs to compute each cell position inside its parent layout. The basic case is to recover the order in which widgets are added into their parent layout. As an example, in a VerticalFlowLayout or HorizontalFlowLayout, the order in which widgets are added controls their position one relative to the other, see for example Listing 4.2 (lines 6 and 7) where two widgets are added consecutively to their parent.

Spec having a limited number of simple layouts (VerticalFlowLayout and HorizontalFlowLayout), it falls within this easy case.

```
1   int row = 0;
2   Grid grid = new Grid();
3   grid.setWidget(0, 0, new Label("name:"));
4   grid.setWidget(row++, 1, new Button());
5   grid.setWidget(++row, 1, new Label(""));
6   grid.setWidget(grid.getRowCount(), 0, new Label());
7   grid.getFlexCellFormatter().setWidth(0, 1, "50%");
```

Listing 4.4: Complex layout creation in Java GWT

More complex layouts, like the Grid layout, allow cells to occupy (span) several

positions in the grid or to be inserted in any position of the grid. For GUI based on markup languages (*e.g.*, GXT, HTML), position computation is still relatively easy as the information is hardcoded in the source. For GUI based on programming languages (*e.g.*, Java GWT), the position or span might be the result of computations at execution time and, therefore, more challenging to extract. For example see lines 3 to 6 in Listing 4.4.

We try to solve some of these cases by performing symbolic computation. The same ideas were used in other GUI extraction tools ([Silva et al., 2010], see Section 2.2.4). Symbolic execution involves identifying all functions/operators' semantics that can be used in the source code. Here, in the example of Listing 4.4, one needs to know the Pre/Post Increment/Decrement operators (lines 4 and 5) as well as the getRowCount, getColumnCount, getCellCount methods, and the assignment operation (line 1). Concretely for our examples, we only need the pre and post-increment operators (x++ ; ++x), and the getRowCount and getCellCount methods to perform the computation. Using symbolic computation, we can resolve the successive values of row in Listing 4.4.

4.2 Visual code generation

Once we have a GUI model, it is possible to generate the GUI in the target framework. As for the extractors, we implemented three generators for three different frameworks. The first one, Angular, is web-based, and the interface is defined in a markup language (HTML). The second one, Seaside, is web-based, and the interface is defined in a programming language (Pharo). The last one, Spec2, is desktop-based, and the interface is defined in a programming language (Pharo).

When generating the target code, it is possible to use an intermediate target meta-model or to embed the target meta-model inside the Pivot meta-model to target language mapping (see Section 3.2.1). In these implementation examples, we chose the second option, *i.e.*, no target meta-model.

In this section, we discuss the steps presented Section 3.3.3 to generate the target code with concrete examples from the generators (see Figure 3.14, *Visual code generation*). We present, again, separately the sub-steps to support a new framework (see Section 4.2.1) and the sub-steps to migrate a new application (see Section 4.2.2) with concrete examples from the generators.

4.2.1 Visual code generation — framework configuration

To handle an application using a new framework, one needs to:

- Identify target framework environment, and

- Map Pivot meta-model to target framework

Identify target framework environment. Before working on concrete code generation, it is essential to discover the architecture required by the target GUI framework. The simplest solution is to read the target framework's documentation to understand the good practices when developing an application with this framework. We also define how to import GUI dependencies in the target framework, *e.g.*, how to import modules in Angular. In Angular, the GUI code is defined inside an HTML file. However, it is also necessary to create several configuration files (*e.g.*, module, CSS, route). All those files are identified at this sub-step and are necessary to create the target GUI. In Spec2 and Seaside, the GUI definition code has to be written in a specific method.

Map Pivot meta-model to target framework. As described in Section 3.2.3, our migration includes the mapping between source frameworks and our Pivot meta-model and between our Pivot meta-model and the target framework. This step takes care of the second dictionary mapping. Building the dictionary that maps pivot meta-model concepts to the target framework is similar to building the dictionary mapping the source framework to the Pivot meta-model. In the case of Angular, we map pivot widgets to Angular tags (*e.g.*, a Button corresponds to <input type="button"/>). In the case of Spec2, we determine the methods used to instantiate the target widgets. For instance, the common widgets (*i.e.*, button, label) should be instantiated using a factory pattern. In contrast, the traditional call to a constructor method is favored for less frequent or more complex widgets. Finally, the Seaside framework uses invocations to factory methods to build the widgets. As for the extraction, a pivot meta-model concept can correspond to multiple widgets in the target framework.

4.2.2 Visual code generation — application generation

One also needs to perform two sub-steps when it comes to generate the target code of an application using a known GUI framework:

- Identify target application environment, and
- Generate code

Identify target application environment. This sub-step consists of determining the environment in which the target application will be executed. The sub-step is identical for GUI defined with markup and programming languages. It configures the endpoint URL for data access and URL to retrieve images (*e.g.,*). The generator must use this information to create a runnable application in the target environment. The approach will complete the target GUI

code with configuration files (or annotations for programming languages) for all our generators.

Generate code. The code generation consists of creating the target GUI. The main approach is to visit the pivot DOM and, for each widget, create its counterparts in the target application. In the case of Angular, the generation is eased because the generated HTML file and the pivot DOM have the same structure. For Spec2 and Seaside, our generator creates into specific methods the widgets and their composition using constructors and a pre-defined set of methods. For Spec2 specifically, the DOM must be defined in a method and the widgets instantiated in another method.

4.3 Behavioral code extraction

In the following, we present an implementation of our behavioral migration approach, presented in Section 3.4 (see also Figure 3.14, *Behavioral code extraction*) only for the migration of Java GWT Behavioral code. As the approach is split into two parts, we split the implementation into two parts: configuring new framework extraction, Section 4.3.1, extracting application Behavioral code, Section 4.3.2. We also present the critical information one should pay attention to for adapting our extractor for other programming languages in Section 4.3.3. Note that, again, as described in Section 3.4.2 and in Section 3.4.3, sub-steps are divided into supporting events and the Manipulation code extraction.

4.3.1 Behavioral code extraction — framework configuration

We are now detailing the two sub-steps to configure a new framework for Behavioral code extraction:

- Identify event handler types, and
- Define Manipulation code patterns

As presented in Section 3.4.2.1, the first sub-step is dedicated to the configuration of event extraction and the second sub-step to the configuration of Manipulation code extraction.

Identify event handler types. To identify all event handler types, we looked at the GUI framework documentation. The GWT documentation[3] defines the class EventHandler as the most abstract event handler type. Thus, the available event handlers in the source application are the subclasses of EventHandler.

[3] http://www.gwtproject.org/javadoc/latest/

Define Manipulation code patterns. To detect Manipulation code inside the source code model, we manually defined patterns that recognize Manipulation code. In the following, we present the eight patterns we defined for the six pieces of Manipulation code of our Behavioral code meta-model (see Section 3.4.1).

For `OpeningPopup`, we defined three patterns. The first one is about looking in the AST for a reference to a Java class named ErrBox. The second one is similar to the first one, but looking for a Java class named EventPopup. An example of AST matched by this pattern is presented in the *Model transformation* sub-step, Section 4.3.2, and depicted in Figure 4.1, page 67. The third one is about looking for a method invocation. The method invoked must be named alert, and the receiver of the invocation is a reference to a Java class named Window. An expression matching the pattern is Window.alert(...).

For `Navigating`, we defined one pattern. It looks for a reference to a class named Workspace. This class should be the receiver of an invocation of getPhaseManager() which, in turn, is the receiver of an invocation displayPhase(). This pattern matches the Workspace.getPhaseManager().displayPhase(...) expression. Again, an example of AST matching by this pattern is presented in Figure 4.1.

For `OpeningDialog`, the pattern matches an invocation of a method named show. The receiver of this invocation must be a local variable reference. Moreover, this local variable must hold a Dialog widget. At this stage, we remind the reader that we extracted the variables to which widgets are assigned during the Visual code extraction. This pattern matches the dialog.show(...) expression.

For `AccessingDOMElement`, the pattern matches a reference to a variable that contains a widget.

For `GettingAttribute` and `SettingAttribute`, the patterns looks for already matched `AccessingDOMElement` Manipulation code. Then it checks that this `AccessingDOMElement` is the receiver of a getter or a setter method invocation. Getter or setter methods are simply methods with their name beginning with "get" or "set". The name of the attribute is retrieved from the name of the getter or setter, *i.e.*,setTitle() corresponds to a setter of the attribute *title*.

4.3.2 Behavioral code extraction — application extraction

Once an extractor is configured for a GUI framework, it can extract Behavioral code from any application using this GUI framework. To do so, it performs four sub-steps; the first two are dedicated to events extraction and the last two to Manipulation code extraction (see Section 3.4.2.2).

We now detail the implementation of the two sub-steps for events extraction:

- Detect event handler instances, and
- Attach event handlers to widgets.

Listing 4.5 presents a snippet of code that illustrates event handlers creation in Java. The code consists in the creation of three widgets: line 1 a panel, line 2 a linkbutton, and line 5 an anonymous button (new Button()).

```
1 Panel panel = new Panel();
2 LinkButton linkbutton = new LinkButton("Send");
3 linkbutton.addClickHandler(new ClickHandler() {
4   public void onClick(ClickEvent event) { ... }});
5 panel.add((new Button()).addClickHandler(new
     ClickHandler() { ... }));
```

Listing 4.5: Creating event handlers in Java/GWT

In Listing 4.5 only the ClickHandler type is represented (lines 3 and 5).

Detect event handler instances. The basic approach consists of looking for the constructor invocations of the event handler types. For instance, creating a `Click` event in our pivot model (see Figure 3.11) is made by calling new ClickHandler(...).

In Listing 4.5, there are two event handler creations, line 3 and 5, both identified by new ClickHandler.

Attach event handlers to widgets. For each handler instance, our implementation extracts its widget owner. To do so, it looks for the receiver of the handler's creation. The extractor performs an analysis of the source code model to retrieve the widget owner (see Section 4.1.2). The owner can be declared in the same method or another method or class.

In Listing 4.5, the first click handler (line 3) is created inside the method addClickHandler sent to the variable linkbutton. And the variable linkbutton holds the linkbutton widget defined line 2. Thus, the event handler owner is the LinkButton widget. The second click handler (line 5) is created inside the method addClickHandler sent to the anonymous instance of the Button class. Thus, the event handler owner is the anonymous button.

In addition to the extraction of the events, our approach extracts Manipulation code from the application. The extraction of Manipulation code is done in two sub-steps:

- Apply Manipulation code patterns, and
- Manipulation code model transformation.

Listing 4.6 presents a snippet of code that includes Manipulation code. In this example, the method onClick() is called when the end-user clicks on a button which, in turn, calls the method generateError(). In case the application has

been launched in debug mode (line 5), a Popup is displayed with the message "I am an error" (line 7). Otherwise, the navigation to the root APage is performed (line 9).

```
1 public void onClick(final ClickEvent event) {
2   this.generateError();
3 }
4 private void generateError() {
5   if(debugMode){
6     System.err.println("logging error");
7     EventPopup.displayError("I am an error");
8   } else {
9     Workspace.getPhaseManager().displayPhase(
     ConstantsPhase.APage());
10  }
11 }
```

Listing 4.6: Example of Manipulation code

Apply Manipulation code pattern. To extract Manipulation code, we first use a pattern matcher on the source code model (see Figure 3.14) with the pattern defined at the configure framework level (see Section 4.3.1). This step retrieves Manipulation code position inside the source code model. In our example (Listing 4.6), our implementation identifies two pieces of Manipulation code. EventPopup.displayError(...) matches one of the `OpeningPopup` pattern, and Workspace.getPhaseManager().displayPhase(...) matches the `Navigating` pattern.

Manipulation code model transformation. Then, we perform model transformations on detected Manipulation code. Figure 4.1 presents the model transformation performed for the method generateError() of Listing 4.6. The left-hand side presents a simplified version of the source method AST. The circled entities (Worspace, getPh..., EventPopup, displayError(), and "I am an...") are the entities found by the pattern matcher. The right-hand side presents a simplified version of the produced pivot model. For instance, for the `OpeningPopup` (Listing 4.6, line 7), we replace EventPopup.displayError by a `OpeningPopup` entity. The string parameter is preserved during the transformation. The case of `Navigating` is more complex, as the parameter ConstantsPhase.APage() refers to a `Root` defined in the UI model, our approach also retrieves the root (in gray in Figure 4.1).

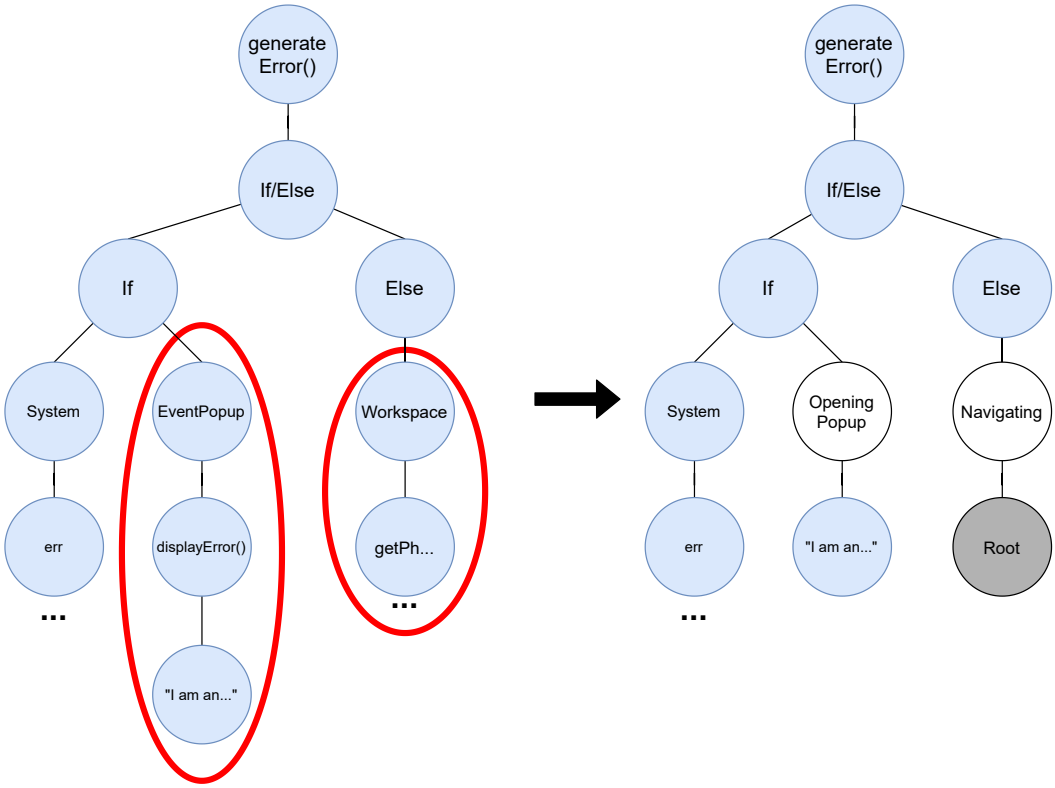


Figure 4.1: Example of model transformation for Listing 4.6
left: Source AST ; right: GUI pivot model
□ Concepts of the behavioral package
■ Concepts shared with the Visual code packages
□ Concepts shared with the generic AST meta-model

4.3.3 Behavioral code extraction — adapting to other programming languages

We presented the implementation of our Behavioral code extractor for the Java/GWT programming language. To help future research adapt our work for the migration of other applications, we now present some lessons and tools.

Our extraction approach is based on pattern matching on an AST. Designing an AST and parser for a source language is time-consuming and error-prone. Thus, we advise using an existing tool when possible. Even if they are defined using an unknown programming language of the practitioner, the effort of learning the language is worth it. In our context, we used the FAST meta-model, which is an abstract meta-model that can be extended to represent different programming languages ASTs. We also use the SmaCC project [Brant et al., 2017] that comes with several parsers.

Another takeaway message is to focus on the main pieces of Manipulation code and events. As depicted in Section 3.1.2.2, hundreds of events and kinds of Manipulation code exist. Thus, building a migration tool that deals with all of them for industrial projects is not feasible. We advise future researchers to perform a first analysis of the source code to determine the Manipulation code and events of interest for the migration.

Finally, as the extraction is based on pattern matching, the higher the number of patterns is to be defined, the worst it is. To avoid creating many patterns, one should first perform a source code standardization [Włodarski et al., 2019]. It consists of modifying the source code to express Manipulation code with as few different expressions as possible. Thus, only one pattern is to be defined for each piece of Manipulation code.

4.4 Behavioral code generation

From the behavioral model, it is possible to generate the target code, which is the last step of our approach. As depicted in Figure 3.14, *Behavioral code generation*, we split the generation into two parts: Section 4.4.1 presents how we configure our approach to support new framework, Section 4.4.2 presents how we actually generate events and Manipulation code. We also implemented additional features to ease the migration (Section 4.4.3).

4.4.1 Behavioral code generation — framework configuration

Before performing the actual generation of the code, we configure our approach to support a new target framework. During this step, we build the mapping between

the pivot events and the Manipulation code with the target framework. With the mapping between pivot Visual code and target language (see Section 4.2.1), these mappings take care of the Pivot meta-model to target framework mapping (see Section 3.2.1). Two sub-steps are necessary to create the Behavioral code generation mapping:

- Map pivot events to target framework, and
- Map Manipulation code to target framework

The first step builds the mapping between the events and the target framework.

Map pivot events to target framework. The basic approach consists of creating a dictionary. In the case of Angular, we map the `Click` concept (see Figure 3.11) to the Angular attribute (click), or the `Submit` concept to the Angular attribute (ngSubmit).

Then, the second sub-step to support the generation of Behavioral code using a new framework consists of mapping Manipulation code with the target framework.

Map Manipulation code to target framework. To map Manipulation code to its counterpart in the target framework, we create a dictionary. This mapping includes for each piece of Manipulation code: its textual representation and its dependencies (*e.g.*, import statements and variable initializations).

4.4.2 Behavioral code generation — application generation

Once we configured our approach for a new framework, it is possible to actually generate the target Behavioral code for a given application. To illustrate the Behavioral code migration, we present the migration of the Java piece of code presented in Listing 4.7 to Angular.

The Behavioral code generation is split into two sub-steps:

- Generate event handlers, and
- Generate target AST.

The first step consists of the generation of the event handlers in the target code.

Generate event handlers. During the generation step of the Visual code (Section 4.2), our tool adds handlers instances to the list of widgets attributes. Listing 4.7 line 4, the link button has a click handler. In Angular, the handlers are defined in the HTML file representing the Visual code using the *event-binding* feature. Figure 4.2 presents how event-binding feature is used. It consists of adding in the HTML source code the template statement (*e.g.*, a method) executed when an event is fired.

```
1 Panel panel = new Panel();
2 LinkButton linkbutton = new LinkButton("Send");
3 panel.add(linkbutton);
4 linkbutton.addClickHandler(new ClickHandler() {
5     public void onClick(final ClickEvent event) {
6     String values = emailBox.getText();
7     if (values != null) {
8       values.split(",");
9       EventPopup.displayInfo("can access");
10    } else {
11      Workspace.getPhaseManager().displayPhase(
      ConstantsPhase.AnotherPage());
12    }
13  }});
```

Listing 4.7: Example of Java code

Figure 4.2: Angular Event Binding feature

```
1 <panel>
2   <input type="email" name="emailBox">
3   <button (click)="onClick()">
4     Send
5   </button>
6 </panel>
```

Listing 4.8: Example of HTML code migrated from Java code in Listing 4.7

Listing 4.8 present the generated HTML file for the Listing 4.7 migration example. Line 3, we generate the Angular (click) attribute that is linked to the template statement onClick() when end-users click on the button.

To finalize the migration, our tool generates the code executed when an event is fired. It includes the generation of the target code based on the pivot AST (see Section 3.4.1). Manipulation code is generated following the mapping defined in Section 4.4.2.

Generate target AST. We generate the target code in the target language based on the Behavioral code model. The generation is done by visiting the model and generating its target language counterpart for each node. Here we remind the reader that the Behavioral code model is comparable to a modified AST (see Section 3.4.1).

For the Manipulation code, we also generate its dependencies. For instance, the Angular navigation service is required to perform the navigation from one page to another. To generate the code that initializes Manipulation code dependencies, we used the mapping between Manipulation code and its dependencies defined in the preceding section. Then, during the code generation step, our implementation generates the code to initialize the dependencies in the class constructor. In our context, it is the case for multiple Manipulation code elements. For instance, the `Navigating` and the `OpeningPopup` Manipulation code need to use two Angular services: DesktopService, and ToastrService.

```
1 constructor(
2   protected _desktopService: DesktopService,
3   private _toastrService: ToastrService,) {
4 }
5
6 onClick() {
7   let values = (<any>this.input).nativeElement.value;
8   if (values != null) {
9     values.split(',');  // <ToReview> : Unknown
     invocation: split(...)
10    this._toastrService.success('can access');
11  } else {
12    this._desktopService.openPage('AnotherPage');
13  }
14 }
```

Listing 4.9: Example of TypeScript code migrated from Java code in Listing 4.7

The Listing 4.9 present the generated TypeScript file for the Listing 4.7 migra-

tion example. Lines 2 to 3, the generator automatically declared the DesktopService service dependency to allow the navigation between pages and the ToastrService used by the OpeningPopup Manipulation code.

4.4.3 Additional features

In addition to the Behavioral code migration, we have implemented features in the generator to help developers during the migration process. It consists of making the code more natural (*i.e.*, respecting the conventions of the target language as used by an expert).

Migration of Java to TypeScript. Additionally to the migration of the Manipulation code, our implementation migrates the rest of the code (*i.e.*, variable declaration, control flow, *etc.*). Although it is not one of our main goals, it helps developers understand the target language[4] and speeds up the migration process. For instance, Listing 4.7 line 6, the variable values is a string declared in Java. Our generator produced, Listing 4.9 line 7, let values which corresponds to the values variable declaration in Angular.

Add comments. Our generator adds comments with a tag *ToReview* at the end of each statement where part of the statement is not fully migrated. It helps the developers focus on problematic expressions. For instance, Listing 4.7 line 8, the split(...) method is not known by our tool, so, it is migrated as a TypeScript method invocation, Listing 4.9 line 9, and flagged with the comment Unknown invocation.

Follow target framework guidelines. The generated code should use the target framework features. Indeed, it is important to follow the target framework guidelines to produce natural code. For example, we can use both JQuery or the Angular framework to access a DOM element in our context. Whereas the code using JQuery would be more concise, we prefer to use Angular native features. Listing 4.9 line 7, (<any>this.input).nativeElement is an Angular DOM element access. Using JQuery, the code would be translated as $("#input"), which is more concise but does not use Angular features.

```
1 <div>
2   <img [src]="itemImageURL
    ">
3 </div>
```

Listing (4.10) Data Binding - HTML part

```
1 class myComponent {
2     itemImageUrl = '../
    path/to/image.png'
3 }
```

Listing (4.11) Data Binding - TypeScript part

Figure 4.3: Data Binding in Angular

[4]Note that, in our company, most developers are experts in GWT but novices in Angular

We acknowledge that we could use the Angular *data binding* feature to access the DOM elements' value. However, using this specific feature is more challenging because it requires modification of the HTML source code. Figure 4.3 presents an example of data binding of the itemImageUrl attribute. The value of the data is specified in the TypeScript file (Listing 4.11, line 2) whereas the attribute is linked to the GUI in the HTML file (Listing 4.10, line 2).

Allow API switching. API differences exist between the source GUI framework, the target GUI framework, and the GUI meta-model. This problem was raised in the literature (see Section 2.2.2). It consists of using different terminologies in different GUI frameworks to express the same concept. For example, the attribute text in the GUI meta-model represents the content of an input. However, in Angular, it translates as value.

Our generator must take into account the differences to produce code that has the correct behavior. To handle such differences, we manually mapped each source UI concept to its target counterpart. In a concrete example, Listing 4.7 line 6, emailBox.getText() allows one to get the value of the input text emailBox. We manually map the GWT attribute text to the Angular attribute value. So, Listing 4.9 line 7, our tool generated in Angular an access to the value property.

4.5 Conclusion

In this chapter, we presented implementation examples of our approach for the migration of the Visual code and the Behavioral code.

For the Visual code migration, we detailed the extraction sub-steps in Section 4.1 and the generation sub-steps in Section 4.2. For each sub-step, we proposed adaptation for GUI defined using programming and markup languages.

For the Behavioral code migration, we also detailed the extraction sub-steps in Section 4.3 and the generation sub-steps in Section 4.4. We presented the implementation for the migration from Java GWT to Angular. It includes details on the implementation and the mapping between the two frameworks through our Pivot meta-model.

Based on these real implementation examples, we evaluate our approach for the GUI migration in the following chapter.

CHAPTER 5

Migration Validation

Contents

5.1 Visual code migration validation 75
5.2 Behavioral code migration validation 83
5.3 Discussion 87
5.4 Threats to Validity 90
5.5 Conclusion 92
5.6 GUI migration conclusion 93

In this chapter, we validate our approach and the Pivot meta-model. To do so, we perform application migrations using our approach implementations presented in the preceding chapter. We present the validation of the Visual code migration and of the Behavioral code migration respectively in Section 5.1 and Section 5.2. In Section 5.3, we discuss our GUI migration results. In Section 5.5, we conclude on the validation of our approach. Finally, in Section 5.6, we conclude the Part I on the presentation of our approach to support GUI migration.

5.1 Visual code migration validation

We validated the Visual code migration on five real applications. In Section 5.1.1, we present the case studies. In Section 5.1.2, we present the metrics used for the validation. In Section 5.1.3 and Section 5.1.4, we detail the result get for the extraction and the generation.

5.1.1 Case studies

To validate our approach, we migrated five applications: Kitchensink, PostOffice, Traccar, DBManager, and SpecDB. In the following, we present the migration projects with their source GUI framework and target GUI framework. Table 5.1 details the different migrations projects.

Kitchensink and PostOffice are written using BLCore, a web-based GUI defined using programming language, and migrated to Angular, a web-based GUI

Table 5.1: Case study Description

Project	Source Framework	GUI definition Type	Target Framework	GUI definition Type
Kitchensink	BLCore	programming	Angular	markup
PostOffice	BLCore	programming	Angular	markup
Traccar	GXT	markup	Seaside	programming
DBManager	Spec	programming	Spec2	programming
SpecDB	Spec	programming	Spec2	programming

defined using a markup language. BLCore is the custom GUI framework of Berger-Levrault that extends the GWT GUI framework with specific widgets. This framework consists of 763 classes in 169 packages. It also encourages some coding conventions. Angular is a modern GUI framework supported by Google based on TypeScript.

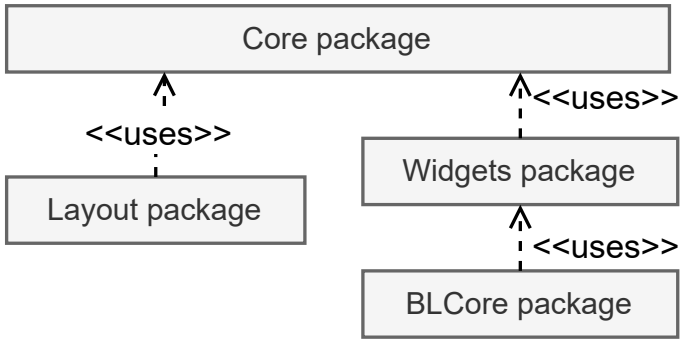


Figure 5.1: BLCore - GUI meta-model

The GUI meta-model used for BLCore GUIs extraction, presented in Figure 5.1, is composed of the Visual code packages (Section 3.3.1): the layout package using the core package (Figure 3.8), and the widget package (Figure 3.7) that extends the core package. We also extended the widget package with a BLCore package. This package includes eight widgets created by Berger-Levrault. It allows us to deal with the Custom Widget (see Section 3.3.1.4) of the company.

Kitchensink is a closed-source application of Berger-Levrault. This software system targets developers and gathers the widgets available for building a user interface at Berger-Levrault. It contains 470 Java classes and 56 web pages. PostOffice is a software system used in French administrations. It aims to ease the dispatch and digitalization of (ground) mails. It contains 3,227 classes and 98 pages.

Two migration cases, DBManager and SpecDB, use Spec as a source framework and Spec2 as a target framework. SpecDB is part of the Spec widgets presentation package. It is used to show the different configurations of a Spec button. DBManager[1] provides a GUI to manage the connections between Pharo and databases. Its user interface is divided into multiple pages. Note that even if Spec2 is the next version of Spec, the framework has been completely rewritten. So the migration corresponds to a GUI migration and not a GUI framework update.

The last one, Traccar[2], uses the GXT framework (a GWT extension) as a source framework and Seaside as a target framework. Traccar is an open-source server and web client for various GPS tracking devices. It contains 649 classes and 28 pages.

We now give more detail about the GUI of each application to migrate. For instance, we provide the number of widgets, attributes, and roots (*i.e.*, number of web pages or desktop windows) to extract and generate in the target application. Table 5.2 gathers the information about the different projects.

Table 5.2: Application descriptions

Source framework	Project	Widgets	Attributes	Roots
BLCore	Kitchensink	238	156	6 *(out of 56)*[1]
	PostOffice	724	1065	10 *(out of 98)*[1]
GXT	Traccar	125	104	3 *(out of 28)*[1]
Spec	DBManager	38	27	3
	SpecDB	15	21	1

[1] 10% sample

For the Kitchensink, PostOffice, and Traccar projects, we take a sample of the root pages for validation. So we present the number of widgets, attributes, and roots of the sample. This choice is explained in the next section (Section 5.1.2). There are for Kitchensink 238 widgets and 156 attributes; for PostOffice 724 widgets and 1065 attributes; for Traccar 125 widgets and 104 attributes; for DBManager 38 widgets and 27 attributes; for SpecDB 15 widgets and 21 attributes.

5.1.2 Validation set-up

We divided the migration validation into two parts: extraction validation and generation validation. The extraction validation consists of checking that our pivot

[1] https://github.com/juliendelplanque/DBConnectionsManager
[2] https://github.com/traccar/traccar-web

model includes the elements of the source application. It compares the DOM and the attributes of the GUI without considering the final visual aspect. The generation validation consists of visually comparing each source page with the exported one.

For the **extraction validation**, we check that all the widgets and attributes are detected and correctly identified. This validation consists of the following three metrics [Hayakawa et al., 2012, Joorabchi and Mesbah, 2012, Sánchez Ramón et al., 2014]:

- The percentage of widgets correctly *detected* regardless of whether their types are detected or attached to the correct container and created in our pivot model. It checks that the number of widgets in the source application is the same as in our pivot model.
- The percentage of widgets correctly *identified*, *i.e.*, a button in the source framework corresponds to a button in the pivot model. On the contrary, the widgets not identified are mapped to `Custom Widgets` or wrong widget types (*e.g.*, a button mapped to a panel).
- The percentage of widgets *assigned* to the correct container. It validates the DOM building.

We use the same metrics for the attributes:

- The percentage of attributes correctly *detected* compared the number of attributes in the source application with the number of attributes in our GUI pivot model. This checks that we identified all instantiations of widget types, even for Custom Widgets (unknow types) and calls to factory methods.
- The percentage of widgets correctly *identified* validates that each source attribute is correctly mapped to its attribute concept counterpart.
- The percentage of attributes *assigned* to the correct widget checks that each attribute is attached to the same widget in the source application and in our GUI pivot model.

We rely on manual validation to check all these metrics. Because the manual validation is tedious and error-prone for large applications, we take a sample of the pages of the Kitchensink, PostOffice, and Traccar applications. For each case, we consider a sample representing at least 10% of the application. We randomly selected 6 pages out of 56 for Kitchensink, 10 pages out of 98 for PostOffice, and 3 pages out of 28 for Traccar. The sample selection is further discussed in Section 5.3.2.

For the **generation validation**, work has been proposed to compare images [Moran et al., 2018, Cao et al., 2010] of the source GUI with the migrated one. However, none is directly applicable to our migration cases. Indeed, to apply this strategy, one must deal with several challenges [Bragagnolo et al., 2020b]:

- *Page access*: to automatically take the screenshot of the pages, one needs to access every page of an application. This step is easy for standard web applications in which all pages are accessible using their URL. However, for applications in which a page is accessed only when a user performs some interaction, such as clicking on a button, getting access to every page is tedious. It is the case for desktop applications and for web applications using an Ajax-based architecture.

- *Successive shifting*: when a migrated widget is not visually equivalent to its original version, it might impact the position of others widgets. A few pixel differences can add up and give a migrated page completely different from the source one.

- *Dynamic content support*: some widgets, as a table, display information coming from an external server. The same data would need to be presented in the source and target generated application for the validation to be successful.

Validation by image comparison is further discussed in Section 5.3.3. One could also think of using tests for validation. However, this is not applicable in our cases because the migrated applications do not have such tests.

Thus, we rely on a manual visual comparison of the pages. First, we check that the generated application is runnable. Then, we visually compare the application with the source one.

5.1.3 Extraction result

We perform the extraction on the five case studies projects. In this section, we report the extraction result using our tool. Table 5.3 summarizes the extraction results.

Our tool detects 99% of all the widget instantiations for all the applications. The six non-detected widgets are created with a factory not present in our widget to pivot concepts map. To solve the missing widgets detection, one needs to add the corresponding factory methods in the mapping source framework to Pivot meta-model mapping (see sub-step *Map source framework to Pivot meta-model* in Section 3.3.2.1). It shows that we have good heuristics to find out the widgets in the source applications.

Table 5.3: Extraction results

Framework source	Project	Widget detected	Widget identified	Widget well assigned	Attribute detected	Attribute identified	Attribute well assigned
BLCore	Kitchensink	100% (238)	94% (224)	99% (236)	77% (118)	95% (112)	100% (118)
	PostOffice	99% (718)	99% (712)	96% (695)	88% (940)	98% (923)	99% (937)
GXT	Traccar	100% (125)	99% (124)	100% (125)	81% (84)	100% (84)	100% (84)
Spec	DBManager	100% (38)	94% (36)	100% (38)	92% (25)	100% (25)	100% (25)
	SpecDB	100% (15)	100% (15)	100% (15)	67% (14)	100% (14)	100% (14)
Average		99%	98%	99%	87%	98%	100%

Our implementation identifies correctly 98% of the widgets type. For the Traccar and DBManager, the tool misses widgets used in toolbars. This kind of widget is mainly used in desktop applications, and since our widgets meta-model comes from W3School, which describes standard web components, we did not have them. For Kitchensink and PostOffice, the tool identifies 94% and 99% of the widgets. All the unidentified widgets are created by the company for its business. So they are mapped to `Custom Widgets`. To solve these problems, one can extend our meta-model with the missing widgets.

Except for the BLCore framework, all the widgets are perfectly assigned to their container. With BLCore, the problem comes from the variety of ways to define widget containment. For instance, the custom widgets defined in BLCore does not use our heuristics, *i.e.* call to method `add()` or parameter in the constructor (see Section 3.3.2.1, *Identify containment rules*), to define the containement. Each custom widget defines its method that defines the containment. One solution to this problem would be first to modify all custom widgets to make them use the `add()` method. Another solution is to add new containment identification rules in our tool.

Attributes are correctly detected at 87%. The best result appears in the DBManager application with 92%. Attributes are harder to detect for two reasons. (1) GUI frameworks define default attributes for widgets, so we have to analyze those attributes and add them in our extractors manually, for instance, the color of buttons is defined at the level of the GUI framework and not for each button instance of the application, and (2) in programming languages, attributes can be declared in multiple ways: using a setter or a parameter in a constructor. This diversity forces us to analyze all the possibilities since we did not find a heuristic that will select all attributes definitions. Again, to avoid enumerating all possibilities in our extractors, it would be beneficial to standardize the code to migrate.

All the attributes of Traccar, DBManager, and SpecDB are identified by our tool. For Kitchensink, 95% of the attributes are identified, and for PostOffice, 98% of the attribute are identified. The attributes incorrectly identified are attributes absent in our Pivot meta-model. One solution to deal with those attributes is to add them in our BLCore package (see Section 5.1.1, Figure 5.1)

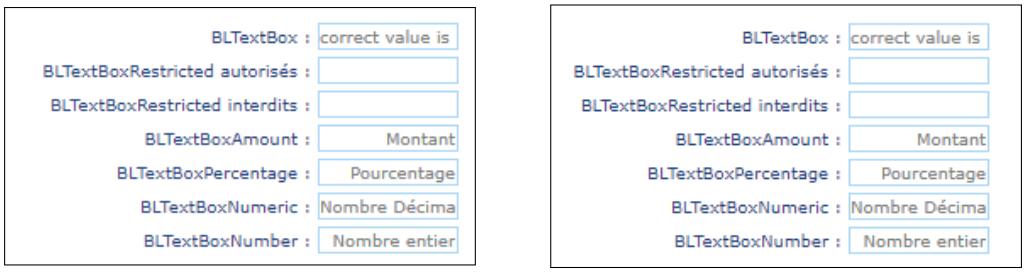
Finally, nearly all the detected attributes are well assigned to their container.

Our implementation of the Visual code extraction approach gives good results for widgets and attributes extraction. It extracts 98% of the widgets and 87% of the attributes. It is possible to improve our extractor by adding specific widgets and attributes concepts to our meta-model and add them to our source framework to Pivot meta-model mapping.

5.1.4 Generation result

Then, we generated the five applications using the target GUI framework with our tool. All the generated applications are runnable out of the box.

We visually compared the aspect of the pages where the widgets are well identified. In the following, we present a comparison for three of the case studies presented in Section 5.1.1. Note that other comparisons for the Kitchensink and the Traccar migration are available on the documentation page of our project[3].



(a) GWT source

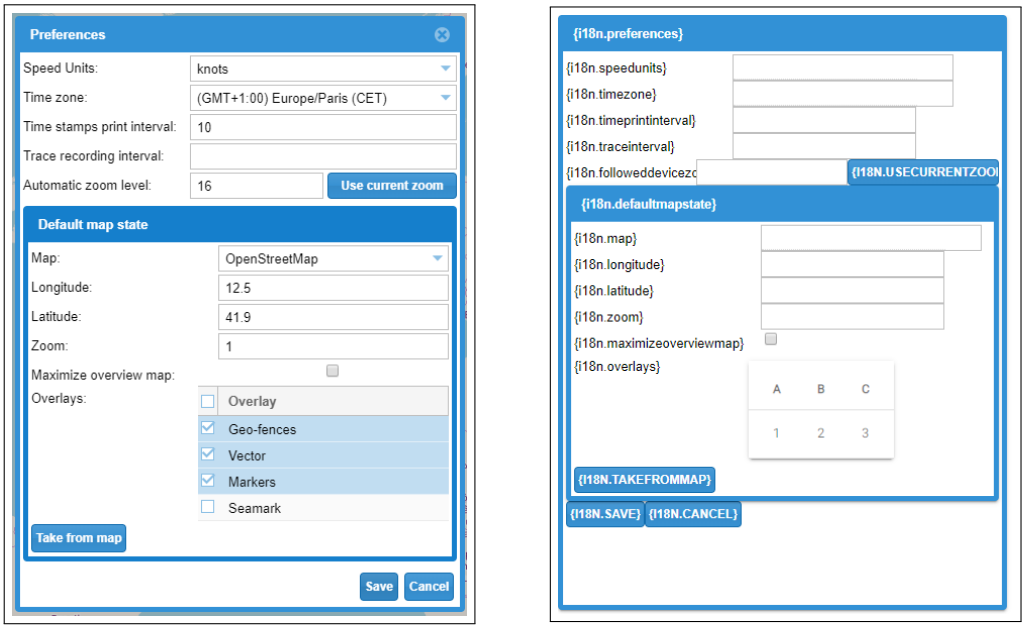
(b) Angular migration

Figure 5.2: Visual comparison of a page migration (Kitchensink)

Figure 5.2 presents the visual differences for the page *Input box* of the Kitchensink application. On the left-hand side, there is the page in the source application, and on the right-hand side, the page after the migration. We can see that there are no differences between the two versions.

Figure 5.3 presents the visual differences for the User Setting page of Traccar. On the left-hand side, there is the page in the source application, and on the right-hand side, the page after the migration to Seaside. There are more differences in this example. The sizes of the input boxes are different, the text of the labels is replaced by the i18n notation, the "overlay" table has a completely different visual aspect, the checkbox is not centered, and the buttons at the bottom are not well placed. The table is not correctly migrated because the table with selection is a custom widget for the tool, and we did not take the time to introduce

[3] https://badetitou.fr/projects/Casino/#current-results



(a) User Setting source (b) User Setting migrated

Figure 5.3: Visual comparison of the User Setting page (Traccar)

it. In such a context, our implementation generates the super type of the concept, *i.e.*, in this example the superclass of TableWithCheckBox which is Table. So, TableWithCheckBox is migrated as a simple table. The i18n notations are correctly extracted. However, we did not implement the support of these notations in our generator. A first step to solve this problem would be to determine how the target framework supports i18n in the *Identify target framework environment* substep (see Section 3.3.3.1). The other differences come from application-specific CSS. Despite all these imprecisions, the structure and the layout of the page are respected.

(a) DBManager source (b) DBManager migrated

Figure 5.4: Visual comparison (DBManager)

Figure 5.4 presents the visual differences for the DBManager application. Again on the left-hand side, there is the page in the source application, and on the right-

hand side, the page after the migration. There are some differences in this example. The text of the buttons on the left of the image is not present. The buttons on the image's right are enabled but are disabled in the source application and are not correctly placed. Apart from the last difference that comes from a developer hack, all the differences are due to attribute extraction problems. Those problems come from missing attributes identification rules in our tool. Finally, the drop-box input has visual differences between the source and the generated application. It is due to a difference in the drop-box implementation in the source and target frameworks.

The migration does not create the same visual aspect. We saw that there are few differences, and the layouts are respected. Most of the imprecisions come from missing attributes or incorrect values. Those problems are easy to fix manually by developers.

In conclusion, we do not report important differences in the visual aspect of pages of the validation. Thus, the migration of the Visual code is successful.

5.2 Behavioral code migration validation

To validate the migration of the Behavioral code, we built another validation. We evaluated our behavioral migration approach on the Kitchensink case study (see Section 5.2.1). We validated our tool with metrics similar to the Visual code validation one (see Section 5.2.2). Finally, we present results for the extraction (see Section 5.2.3), and generation (see Section 5.2.4).

5.2.1 Case studies

To validate the Behavioral code migration, we performed it on two applications. The first one, Kitchensink, was already used for the Visual code migration validation.

Our implementation runs without raising any errors for both applications. However, there is a lack of automatic tools to evaluate the correctness of the produced code. Thus, we performed a manual validation of the Kitchensink application that took us two full-time weeks. Note that performing the same validation for the Human Resources application, *i.e.* the company biggest application comprising more than 50,000 widgets and 21,000 classes, has been estimated by our industrial partner to 5 man/months. For such a migration project, an incremental migration approach and validation are required. We present work relative to incremental migration in Chapter 6.

5.2.2 Validation set-up

Since no behavioral migration approach was found in the literature, we propose a new validation set-up, divided into two parts: check the structure and check the code naturalness.

For the **structure**, our solution checks that the event handlers are correctly detected and migrated. It consists of the following three metrics: *detect*, *identify*, *assign*, similar to the one used in Visual code migration validation (see Section 5.1.2):

- *The percentage of event handlers correctly detected*, *i.e.*, the event handlers are detected regardless of whether their types are detected or attached to the correct widget. For example, the `onDrag` event handler type is not in our meta-model. However, our approach detects that an event handler exists and that the owner widget has not been extracted during GUI extraction.

- *The percentage of event handler types correctly identified*, *i.e.*, a click handler in the source application corresponds to a click handler in our Pivot model.

- *The percentage of event handlers assigned to the correct widget*, *i.e.*, a click handler attached to a button is extracted as a click handler and attached to the same button in the pivot model.

Because no tool performs such an evaluation, we rely on manual validation to check all these metrics. For the *percentage of event handlers correctly detected*, we looked at the source code of the application, file by file, counting the number of created event handlers, and compared it with the number of elements found by our tool. For the *percentage of event handler types correctly identified* and the *percentage of event handlers assigned to the correct widget*, for each detected event handler, we manually checked, in the source code, that its type and owner were correctly extracted by our tool.

For the **naturalness of the code**, we want to check that the generated code respects the convention of code written by Angular developers. We used external tools that check the quality of the produced code:

SonarQube is a well-known open-source tool that analyzes code quality and security. It categories the errors by severities[4]. Blockers have an important impact on the system and are likely to happen; majors have a limited impact and are likely to happen; minors have a limited impact and are unlikely to happen.

[4] https://docs.sonarqube.org/display/SONARqube71/Rules+-+types+and+severities

TypeScript transpiler reports poorly written TypeScript code and potential problems (such as missing classes). It also detects type matching problems.

Codelizer [5] is an Angular linter that reports problems such as using spaces instead of tabs

To avoid bias in the results, we used the default setting of each tool.

We also compare our approach to JSweet, a transpiler from Java to TypeScript. It claims to produce code that compiles and is "programmer-friendly". This is further discussed in Section 5.3.5.

5.2.3 Extraction result

We now check that the source application's event handlers are well detected, associated with the correct `Event` concept of our behavioral meta-model, and linked to the right exported widget.

Table 5.4: Result of manual event handler extraction check

Event handler detected	Event handler type identified	Event handler correctly assigned
100% (232)	98% (228)	95% (221)

In parentheses: number of event handlers

Table 5.4 summarizes the results for the **structure** check. Our manual evaluation reported 232 event handlers created in the Java code.

Our prototype detected 100% of the created event handlers. Among them, it identified 98% of the event handler types. The four event handler types not detected correspond to handlers for events created by the developers, *e.g. custom events*. For example, one of the events is raised when a custom progress bar widget created by the developers is stopped. To support such events, developers would need to create them in the target language and add the newly created events in the behavioral package (see Section 3.4.1) as a kind of Event (as click, change, *etc.*). 95% of the event handlers are assigned to the correct widget. 4 out of the 11 incorrectly assigned event handlers are the unidentified handlers, so our implementation did not assign them to widgets. For the other 7, their owners were not present in the UI model. To summarize, our approach detected 232 event handlers comprising 214 *click handlers*, 5 *change handlers*, 1 *hover handler*, and 1 *out handler*. The remaining 11 event handlers were not correctly extracted.

[5] codelyzer: http://codelyzer.com/

Our approach allows one to extract the event handlers of an application. It also identifies them and assigns them to the correct widget. As for the Visual code, only the *custom events* needs additional work.

5.2.4 Generation result

Then, we generated the two applications using our tool. We validated the **naturalness of the code** on the Kitchensink application migration. Table 5.5 summarizes the results.

Table 5.5: Natural code

Approach	SonarQube (blocker/major/minor)	TypeScript transpiler	Linter (codelyser)
Our approach	520 (0/98/422)	130	367
JSweet	986 (3/566/417)	6,539	21,344

According to SonarQube, our migrated code has 520 errors. It corresponds to 98 “major” problems and 422 “minor” ones. More than half of the major problems (64/98) are “deprecated HTML attribute usage” and “missing table header”. The first one should be avoided to ensure web browser compatibility. Note that fixing these kinds of problem manually is often an easy task, but fixing them in our generator allows developers to fix the problem for all future migration. The second one creates problems with web accessibility. For instance, assistive technologies, such as screen readers, use table headers to provide context to users. Whereas developing an automatic solution that deals with this problem is tedious, one can tune our generator to highlight the locations where developers have to look in the migrated code. For the JSweet migrated version, SonarQube reports 986 errors with 3 “blocker problems”, 566 “major”, and 417 “minor”. More than half of the major problems (492/566) are TypeScript problems with multiline blocks and control flow that might raise issues with non-empty statements. This analysis shows that our approach produces a code of better quality according to SonarQube than the JSweet approach.

The TypeScript transpiler provides information about the number of unknown class usage, *i.e.*, references to classes that do not exist. For our approach, the TypeScript transpiler reports 130 missing classes. They are helper classes and classes that represent data. For the JSweet exported version, it reports 6,539 missing classes. They are helper classes, classes representing data, and widget and behavioral classes (Button, ClickHandler, *etc.*). Indeed, JSweet only migrates the classes of the source application and not its dependencies which include the GUI

framework. Thus, the classes representing the GUI are not migrated. The transpiler does not report critical problems for both approaches. This analysis shows that our approach generates far fewer problems to fix.

The lint, *codelizer*, reports only minor problems due to code formatting. It reports 367 problems for our approach and 21,344 problems for the JSweet exported version. It also reports missing bracket for *if* and *for* statements for the JSweet exported version. Although brackets are not always mandatory, missing brackets might introduce bugs in the application. Tuning the generator to generate the brackets is an easy task. This analysis shows that our approach has a lot fewer problems than the JSweet approach.

Our approach allows one to generate code that has a lot fewer problems than other existing tools.

The approach gives good results for the migration of the Visual code and the Behavioral code. In the following, we discuss the factors that impact our results.

5.3 Discussion

In this section, we discuss our approach and the validation set-up. We first discuss the reason to preserve the visual aspect during the migration in Section 5.3.1. Then, we present how we ensure good representativity of the sample selected for the validation in Section 5.3.2. We detail the challenges of validating migration using image comparison in Section 5.3.3. We present the manual work required to implement our approach in Section 5.3.4. Finally, we discuss the choice of using JSweet as another migration tool for the validation in Section 5.3.5.

5.3.1 Similar visual aspect

Our approach allows one to migrate the Visual code among different GUI frameworks. As validation, we proposed to compare the visual aspect of the former GUI to the generated one. However, widgets do not have the same visual aspect in different GUI frameworks. For instance, an AWT button does not look like an AngularJS button.

It is possible to perform an expensive manual step to tune the visual aspect of all target framework widgets to mimic the former visual. Because of its cost, performing this step is not always desirable.

Thus, before performing a GUI migration, one should consider the following question: must the migrated application have the exact same visual aspect as the source one, or should it follow target GUI framework visual aspect standards?

Different answers to this question are in the literature. On the one hand, as depicted by Moore et al. [1994], “The resulting user interface should have the true

look and feel of the new environment". On the other hand, for commercial software, keeping the same visual aspect increases the acceptance rate of client users [Sánchez Ramón et al., 2014].

5.3.2 Sample selection

As depicted in the previous sections, we had to validate the Visual code extraction manually. However, performing a manual validation on the 182 pages of the case studies (see Section 5.1.1) is too time-consuming.

Thus, we decided to perform the validation of a subset. To do so, we decided to selected 10% of the pages of the applications. This selection is crucial because it must not introduce bias in the validation. So, to avoid introducing bias, we decided to perform a random selection of the pages. However, it could still not fully represent the application.

To ensure the good representativity of the selected pages, we compare the total number of widgets and attributes in the application to the number of widgets and attributes of the selected pages. The selected pages include 9% of the total number of widgets and 13% of the total number of attributes. The sample also includes 43% of the widget types existing in our Pivot meta-model, including the most common ones, such as button and input, as well as complex ones, such as tab manager and parameterizable fieldset. Thus, it appears that the 10% pages randomly selected are representative of the entire application in terms of the number of elements.

Additionally, to compare the number of elements, we check that selected pages are of different sizes. As a result, the pages have from one to hundreds of widgets. Thus, the randomly selected pages are of different sizes, ensuring no bias in the validation.

5.3.3 Image comparison validation

To validate the proper generation of target GUI, we rely on a manual comparison of the pages visual aspect. We acknowledge the existence of projects that compare the visual aspect of two screenshots [Cao et al., 2010, Moran et al., 2018]. Cao et al. [2010] proposed an algorithm that extracts the GUI layout from a screenshot. Then, one can compare the layout of the source application pages with the one of the migrated pages. Moran et al. [2018] detected differences between two versions of the same GUI after modification made by developers. Since the modifications are minimal and the GUI developed with the same framework, one can expect few differences between the screenshots.

We performed a preliminary work [Bragagnolo et al., 2020b] to adapt image comparison to the validation of GUI migration. In this work, we reported several challenges that must be first solved.

First, for Ajax-based applications such as GWT and Angular, one should be able to browse the pages of the web application. Indeed, to automatically take screenshots of the pages, one must crawl all pages. However, with some frameworks, pages are not directly accessible with their URL. It is the case with some Ajax-based applications, such as the applications using GWT in our context.

Second, we must deal with the successive shifting challenge. When migrating a widget from a source framework to a target framework, it might not have the same visual aspect. However, if a target widget has a size different than the source widget, it introduces a shift in the screenshot of a few pixels. Repeating this shift on several widgets in a page can lead to two completely different source and target pages, preventing an automated screenshot comparison.

Finally, to enable the image comparison in the migration context, one must deal with dynamic content. Indeed, a table filled with data will not take the same space in a GUI as an empty table. However, in our context, we do not support the migration of back-end connections. Thus, migrated tables are always empty, and visual comparison can not be used.

Thus, automatic validation through visual comparison of page screenshots is not applicable in the current state of GUI visual aspect migration among multiple technologies.

5.3.4 Manual work

As explained in Chapter 3 and Chapter 4, the migration requires two manual tasks: (1) mapping the source widgets with our Pivot meta-model concepts and (2) identifying how the DOM is built in each GUI framework.

Mapping the widget consists of analyzing each GUI framework's documentation, retrieving the widgets and their attributes, and mapping them to our meta-model. Whereas the second task is easy for markup languages, it requires more knowledge for programming languages analysis. Indeed, in programming languages based GUI, developers can define the DOM in several ways. So, we had to enumerate all the DOM building possibilities and integrate them into our extractors.

To reduce the required manual effort, one can perform a renovation of its software system before migrating. The renovation consists of improving the source code of the application. To do so, developers reduce the number of code smells and rewrite code to make it easier to manipulate. For example, one can rewrite all widgets creation using the basic `new` method. By following this *Quality First* [Włodarski et al., 2019] rule, the GUI extraction becomes more straightforward, and the DOM is built using only one approach. Widgets are always instantiating using the constructor. Moreover, the number of custom widgets might be reduced in favor of standard ones.

5.3.5 Validation with JSweet

For the Behavioral code migration, we focused on the migration of Java GWT code to TypeScript Angular one. Since we did not find any other tool that migrates Behavioral code, we used JSweet as another Java to TypeScript migration tool. However, JSweet **does not** migrate from GWT to Angular. Thus, we do not have the same goal. Whereas it gives us a baseline to compare to, another study with a tool that migrates Behavioral code would be preferable. However, there is no other tool in the literature that performs such a migration.

5.4 Threats to Validity

This section discusses the validity of our case study using the validation scheme defined by Runeson and Höst [2009]. Threats to construct validity, internal validity, external validity, and reliability are presented.

First of all, our validation does not correspond to a formal validation consisting of research questions and their associated hypothesis testing. We instead performed several experiments with several applications and in different contexts. We want to highlight that defining a formal approach to validate GUI migration should be addressed in future work to improve GUI migration validation quality.

5.4.1 Construct Validity

Construct validity indicates whether the studied measures really represent what is investigated according to the research questions.

Evaluating whether the migrated UI is satisfactory or not depends on different factors. For example, the exact similarity of original and migrated GUI is highly dependent on the effort invested in fine-tuning the CSS. However, this is unlikely to happen in real situations because one would probably prefer to use the look and feel of the new framework (see discussion in Section 5.3.1). We, therefore, resorted to validate whether the GUI was (i) correctly modeled (extracted) and (2) correctly regenerated (from the model). These two conditions are necessary to ensure that all information displayed in the original GUI is available in the migrated one and similarly that all end-user actions originally possible are still available after migration.

Extraction validation We selected three metrics to perform the validation of the extraction: *detect*, *identify*, and *assign*. These metrics were considered relevant in Hayakawa et al. [2012], Joorabchi and Mesbah [2012], Sánchez Ramón et al. [2014]. However, some pieces of information are missing in these metrics. For instance, they do not check that values of widgets' attributes are correctly extracted.

Note that, even if such a missing piece of information is absent from the extraction validation, incorrect attributes values will impact the generation validation.

Generation validation We discuss the generation validation for Visual code and Behavioral code separately.

For the Visual code, we rely on manual visual comparison. This solution allows us to validate the generation from the point of view of an end-user of the application. As discussed previously, it is probably not realistic to want to have the same interface. However, it is a way to ensure that we can regenerate an interface with the same information displayed.

For the Behavioral code, checking code behavior would have required automated tests that do not exist in the company for our case studies (a "normal" situation in the industry) or asking testers to check every piece of code, which would be too time-consuming.

We validated the generated code using linters and SonarQube. We acknowledge that they are not entirely satisfactory as it does not ensure that the generated code compiles or performs the same actions as in the original version.

Instead of ensuring the correct behavior of all the generated methods, we chose to, at least, check the maintainability of the generated code. Thus, we validate that it would be easy for developers to perform potential manual fixes after the migration.

5.4.2 Internal Validity

Internal validity indicates whether no other variables except the studied one impacted the result.

We applied our approach to various applications from different organizations, in different domains, with different source and target GUI frameworks (language and markup frameworks). We are confident that this rules out any bias that the main experiment (GWT to Angular) could have introduced.

The validation relies on manual work performed by the author (see Section 5.3.4). This is always subject to some human mistakes. However, by validating on several applications, including hundreds of elements, human mistakes can not have a large impact on the final result.

The Visual code generation is also manually validated by the author. This may introduce a bias. As discussed in Section 5.3.3, we worked on image comparison validation to reduce this bias. However, the solutions found have proven to be insufficient to validate application migration.

For the validation of the Behavioral code generation, we rely on external tools. We use them with their default configuration to avoid introducing any potential bias.

5.4.3 External Validity

External validity indicates whether it is possible to generalize the findings of the study.

Again, we performed our approach on applications, open and closed source, of different sizes and defined with different GUI frameworks. These GUI frameworks use markup and programming languages. This diversity ensures that our approach can be generalized to several frameworks.

However, we did not experiment with binary (or proprietary) frameworks. For instance, GUI defined with Rapid Application Development (RAD) tools are more difficult to extract. Bragagnolo et al. [2020a] have studied the extraction of the Access projects GUI. In their case study, they had to perform the GUI extraction through the Access IDE, which limited the extraction capabilities, but they were able to populate our pivot meta-model.

Once a pivot model is created, the rest of the approach applies normally. Migration of binary file-based GUI is part of our industrial partner projects with a WebDev to Angular migration project

For the Behavioral code migration, extraction and generation were only validated against closed-source applications of Berger-Levrault. Thus, there is a risk that our results can not be easily generalized. Future replication work must be performed, and it is in the planned roadmap of the company (Access and WebDev applications).

5.4.4 Reliability

Reliability indicates whether others can replicate our results.

Except for the closed-source applications of Berger-Levrault, we provide links to every application used for the validation. We also provide the implementation of our approach as a documented and tested software artifact that anyone can use. The implementation comes with a companion project web page[6] detailing the approach, the results, and direct links to the existing importers and exporters.

5.5 Conclusion

In this chapter, we presented the validation of our approach (see Chapter 3) based on the implementation examples detailed in Chapter 4.

We validated the extraction and generation aspect of both Visual code and Behavioral code. For the extraction, we designed it based on existing proposed validation set-up of the literature (see Section 5.1.2). For the generation, our validation

[6] https://badetitou.fr/projects/Casino/

is based on manual comparison of the visual aspect and comparison of our implementation results with other tools results using well-known quality evaluation software system (*e.g.*, SonarQube, *etc.*).

Then, we discussed our validation set-up and our results. We detailed how keeping the same visual aspect during the migration can be challenging and how we ensured good representativity of the selected samples for the validation. We also presented current challenges when using image comparison to validate GUI migration. Finally, we discussed the manual work one needs to perform to adapt our work and the choice of JSweet as another migration tool.

5.6 GUI migration conclusion

We presented our terminology and concept that composed a GUI. It consists of the Visual code, *i.e.*, the visual aspect of an application, the Behavioral code, *i.e.*, the code executed when end-users interact with the Visual code, and the Business code, *i.e.*, the data manipulated by the GUI. To ease the migration of application GUI, we designed a meta-model representing the visual and the behavioral aspect and an approach based on this meta-model. Our approach comes with several steps and sub-steps that can be tuned to fit different migration contexts. We also proposed ways to perform the sub-steps for markup and programming languages.

Then, we built a tool implementing our approach for different source and target GUI frameworks. We detailed the implementations with both the Visual code and the Behavioral code migration. Following the implementation details we provided, one can easily implement our approach to another context.

Finally, we validated our approach on five migration projects. Some of these projects are part of our industrial context, and others are the migration of open-source applications' GUI. Then, we proposed a validation set-up for both the Visual code and the Behavioral code. It aims to point out the strengths and weaknesses of our approaches.

Whereas our approach gives good results, our implementation does not migrate 100% of the applications' GUI. To finalize the migration, developers must check every page and fix the remaining errors. This process of automatically migrating a page using our tool, manually fine-tuning the migration of this page, and delivering it to end-users is part of our incremental migration approach. In the following, we present how we design this approach, the challenges we had to deal with, and how we actually migrate applications.

Part II

Incremental migration approach

CHAPTER 6

Incremental migration

Contents

6.1 Incremental Migration Approach 97
6.2 Hybrid architecture 99
6.3 Implementation 102
6.4 Conclusion 107

We presented in the previous chapters an approach that enables one to perform the migration of application GUIs. Even if we have shown that the approach is efficient, part of the GUI is not perfectly migrated. To finalize the migration, developers must check all pages and, eventually, fix them manually. When migrating large applications, such as the ones of our industrial partner, this step can last for months. However, the migration team can not spend months finalizing the migration while the clients ask for new developments and modifications.

Consequently, the migration team must use an incremental approach enabling the migration with a hybrid application deliverable to end-users. During the migration period, the hybrid application includes migrated pages written in the new framework co-existing with source pages written in the old framework. Constraints to design such a hybrid application have already been detailed in the literature (see Section 2.3).

In this chapter, we present our incremental migration approach (Section 6.1). This approach uses our GUI migration approach detailed in Chapters 3 and 4 to migrate each page of the source application. Then, the incremental approach consists of integrating each automatically migrated page into a hybrid application delivered to the end-users. The hybrid application is built using our hybrid architecture detailed in Section 6.2. We present an implementation of our hybrid architecture for GWT and Angular in Section 6.3.

6.1 Incremental Migration Approach

An incremental migration approach aims to ensure the delivery process during the migration of an application. To do so, it migrates part of the application, integrates

it inside the source application, and delivers an application including parts with old code and other parts with new code. In the following, we named "hybrid application" the application that includes parts with old code and other parts with new code.

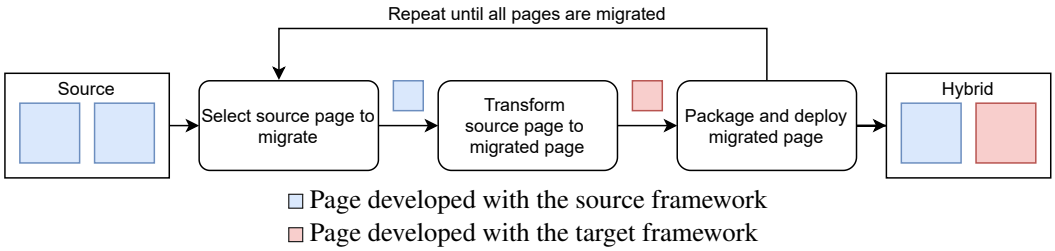


Figure 6.1: The incremental migration process

We designed an incremental migration process that takes advantage of our hybrid architecture. The input of this process is a source application. The intermediate output is a hybrid application composed of migrated pages and the remaining source pages. The final output is a fully migrated application that is equivalent to the source application. This process, illustrated in Figure 6.1, is composed of three main steps as follows:

Select source page to migrate. This step aims to select a page in the source application to be migrated. The migration developers select a page based on their knowledge of the application. Without priority between the page in terms of migration, one is randomly selected. The output of this step is the selected page to migrate.

Transform source page to migrated page. This step aims to transform the source code of the page, selected in the first step, to target language using the target framework. It includes the migration of the three GUI categories of source code: Visual code, Behavioral code, and Business code (see Section 3.1). The transformation of the Visual code and the Behavioral code can be done automatically using our GUI migration approach (see Chapters 3 and 4). The Business code migration is done manually. It consists of the migration of the data transferred between the front-end and the back-end.

Package and deploy migrated page. This step consists of packaging the migrated page. The page is then integrated into the hybrid application to replace the source page. This step results in a hybrid application composed of source and migrated pages.

The three steps of this process are repeated until all source pages are migrated to the target framework. When all pages are migrated, one can deliver the migrated application using the target technology instead of the hybrid application.

Our incremental migration approach is based on the usage of a hybrid architecture. The hybrid architecture allows the integration of migrated pages within the source application. In our industrial context, the GUI frameworks mixed in the hybrid application are GWT and Angular.

This first step, *select source page to migrate*, is not further discussed in this thesis and might be interesting for future research. The second step, *transform source page to migrated page*, was already discussed in Chapters 3, 4, and 5. In our incremental approach, it takes one source page as input and gives the page migrated using our tool as output. The last step, *package and deploy migrated page*, is the focus of the remaining of this chapter. It presents the hybrid architecture in which the pages migrated are integrated.

6.2 Hybrid architecture

Our hybrid architecture allows one to mix, inside an application, two GUIs defined with two different frameworks possibly in different programming languages. In the following, we present the architecture we designed to mix two GUIs and how it is used to build an application. Specifically, it aims to enable mixing GWT and Angular.

Section 6.2.1 presents the hybrid architecture. Section 6.2.2 details how each part of the architecture is compiled to build the final application.

6.2.1 Hybrid architecture description

Our incremental approach enables the migration of applications using a hybrid architecture. This architecture tackles the three challenges depicted in the literature (Section 2.3): *communication*, *type matching*, and *GUI mixing*. For the *communication* challenge, in a hybrid architecture, several frameworks and programming languages are involved. We must define an approach to enable communication between the elements of the hybrid application. For the *Type matching* challenge, when communicating, the elements of the hybrid architecture could exchange data. However, data created in one programing language might not be supported by another programing language. Thus, we have to design allowing data exchange. Finally, for the *GUI mixing* challenge, our hybrid architecture aims to mix GUI defined using different GUI frameworks. Again, a strategy must be designed to integrate widgets coming from one GUI framework with widgets coming from another GUI framework.

Figure 6.2 presents the hybrid architecture in which the front-end is divided into three parts: controller, source pages (*i.e.*, not migrated), and target pages.

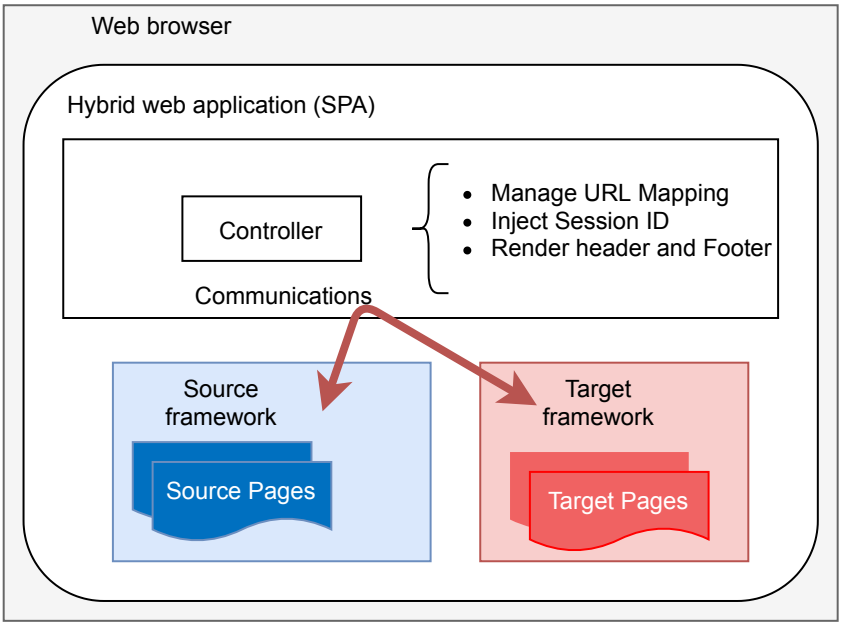


Figure 6.2: Hybrid architecture

The controller is the central part of our hybrid architecture. It can be developed using the source programming language, the target programming language, or another language. It is the front-end entry point of the web application. When the end-users want to access a web page, they request the controller that manages URL mapping to send the correct page. The controller also manages the context of the application (*e.g.*, the currently logged user, the page currently displayed, *etc.*). Because of its central position, it is also in charge of rendering web pages header and footer.

The pages (source and target) contain the GUI visual aspects (see Section 3.1). They are independent of each other, and only navigation between pages is allowed. For instance, a page can not request the value of a text field from another page. In the hybrid architecture, we package pages independently one from the other. A packaged page includes all the code necessary to represent the page GUI, *i.e.* visual, behavioral, and business, see Section 3.1, again. It allows us to integrate the pages inside the architecture independently of their programming language. Thus, packaging pages tackles down the *GUI mixing* challenge.

When navigating from one page to another, the source page might send data to the other one. It is the case when a selected item in an overview page is described in a detail page. To enable such *communication*, the overview page sends the data to the controller. Then, when the data are required, the detail page pulls it from the controller. The central position of the controller is thus used to tackle the *communication* challenge. Moreover, when the data are pulled, the controller checks if the

page that sent the data uses the same GUI framework as the page pulling the data. If pages are defined with different GUI frameworks, the controller takes care of converting data if needed before sending them to the target page. An implementation example of this process is described in Section 6.3.2. Thus, it tackles the *type matching* challenge.

6.2.2 Operational Architecture of hybrid application

Additionally to the hybrid architecture, we designed an operational architecture allowing the pages (source and target) to be compiled into one hybrid application.

One critical challenge is to enable communication between the GUI defined with the source framework and the one defined with the target framework. We identified two main solutions to this problem. First solution, one can use the foreign function interface (FFI) [Polito et al., 2020] that enables one programming language to call methods defined in shared libraries. FFI is a common solution for desktop applications. However, one can not use it for web applications. Indeed, a website accessing user personal files would create a security threat. Second solution, one can compile the source and the target GUIs in the same programming language. This solution requires the source and the target GUI frameworks to be developed with the same programming language, for example, when mixing JavaFX and JavaSwing [Robillard and Kutschera, 2019], or to create transpilers that compile the two GUIs into the same programming language.

Since we are migrating web applications, we selected the second option: transpiling GUIs to the same programming language. We benefit from the Angular transpiler and the GWT transpiler that compiles TypeScript and Java respectively to the JavaScript programming language.

Figure 6.3 presents the operational architecture for a GWT and Angular hybrid application. The hybrid application source code (left) includes the front-end and the back-end parts. The front-end contains the source code of the GWT pages, the source code of the Angular migrated pages, and the source code of the controller. Note that the controller can be developed in any programming language. In our implementation (see Section 6.3), we used the existing GWT controller. The back-end can be developed in any other programming language.

At compilation time (center), we use one compiler for each programming language. Thus, GWT pages are compiled using the GWT compiler, Angular with the Angular compiler, and the back-end with its own compiler (*i.e.*, in our context, the classic Java compiler).

At runtime (right), the compilers produce the hybrid application using our hybrid architecture presented Figure 6.2. In our GWT and Angular example, both are compiled in the JavaScript language to be run in a web browser. The hybrid web application is also linked with the back-end. The back-end does not need to be

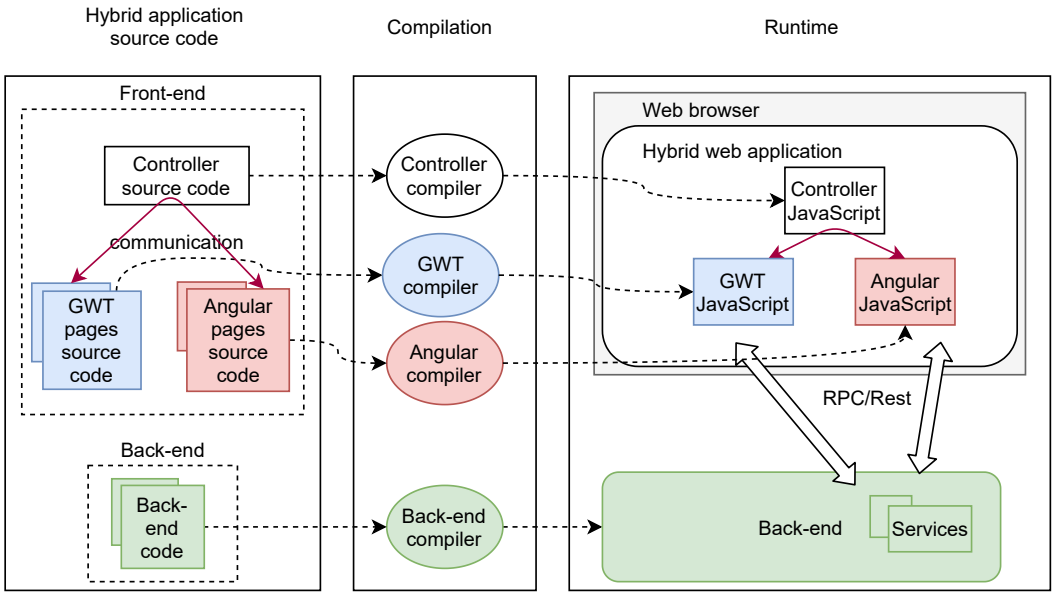


Figure 6.3: Operational architecture of hybrid application for GWT and Angular

migrated. However, we had to perform some modifications in the back-end relative to the front-end to back-end communication. These modifications are discussed in Section 7.4.2.

We presented the main concepts to mix GUI of applications inside a hybrid architecture. Using our architecture, one can migrate the GUI incrementally. Our hybrid architecture tackles the three major challenges: *communication*, *type matching*, and *GUI mixing*. In the following, we detail an implementation for the migration of GWT applications to Angular and detail how it deals with the three challenges.

6.3 Implementation

In the following, we present an implementation of our hybrid architecture (Section 6.3.1) dealing with the *GUI mixing* challenge. Then we describe how we enabled communication in the architecture (Section 6.3.2) to deal with the *communication* and *type matching* challenges.

6.3.1 Implementation of our hybrid architecture through Web Components

Our hybrid architecture mixes independent pages inside one application. The packaged pages must include all the GUI categories of code (Visual code, Behavioral

code, and Business code, see Section 3.1). To create packaged page for the GWT and Angular hybrid architecture, we used Web Components[1]. The idea behind Web Components is to build a reusable JavaScript module from a markup structure (HTML), its associated script (JavaScript), and style (CSS). Then, any Web Component can be inserted into any web application independently of its programming language and GUI framework.

In our context, we build one Web Component from each migrated page produced by our automatic migration tool (see Section 6.1). Then, they are used as packaged pages in our hybrid architecture (see Section 6.2).

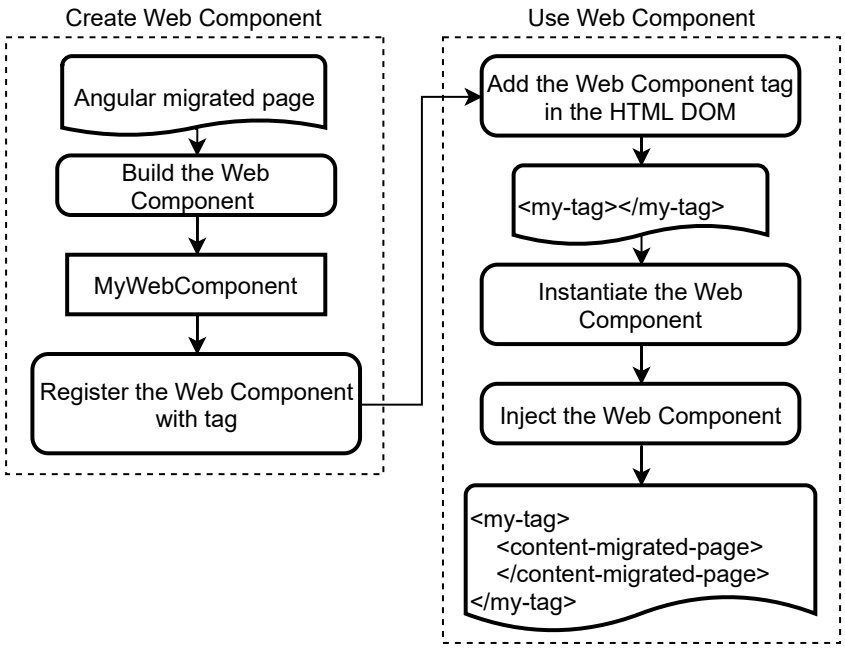


Figure 6.4: Packaging and using Angular migrated page with Web Component

In the following, we present how we concretely package migrated pages as Web Components and deploy them into the hybrid architecture. It consists of five steps. We illustrated the steps to create and use the Web Component in Figure 6.4.

The first two steps, left-side of Figure 6.4, create the Web Component.

1 - Build the Web Component. This step takes as input an Angular page migrated using our GUI migration tool (see Section 6.1) and builds a Web Component. To build the Web Component, we use the Angular element library[2] that can package as Web Component any Angular component. An Angular component is a widget defined by the developer in an Angular application. In our context, each Angular migrated page is an Angular component. In other contexts, they are

[1]Web Component: https://developer.mozilla.org/en-US/docs/Web/Web_Components
[2]Angular element: https://angular.io/guide/elements#how-it-works

similar to custom widgets (see Section 3.3.1.4). Thus, we use the Angular element library on each migrated page to build the Web Components

2 - Register the Web Component. Once we have a Web Component, we register it in the Web Components registry of the web browser. Every web browser has a registry that contains all the Web Components that can be used at runtime. To do so, we use the JavaScript code that registers a Web Component with an associated HTML tag. The HTML tag will then be used to refer to the newly created Web Component.

Once the Web Components are registered, end-users can use the application. The browser performs three steps for each Web Component when end-users navigate to a page: add the Web Component tag in the HTML DOM, instantiate the Web Component, inject the Web Component. These steps are illustrated on the right side of Figure 6.4 They are used to display the migrated pages in the hybrid application at runtime.

3 - Add the Web Component tag in the HTML DOM. First, the end-users use the application. If they navigate to a page with a Web Component HTML tag inside the DOM, a migrated page in our approach, the browser detects the tag and tries to instantiate it.

4 - Instantiate the Web Component. To instantiate the Web Component, the browser looks in its Web Component registry for the registred HTML tag defined in the *register the Web Component* step. From this registry, it retrieves the Web Component and can instantiate it.

5 - Inject the Web Component. Finally, the browser instantiates the Web Component HTML DOM and injects it at the place of the Web Component HTML tag inside the DOM of the navigated page. It also adds the specific styles and scripts of the Web Component. In our context, the injected Web Component contains the migrated Angular page.

Web Components allow one to insert Angular migrated pages inside a GWT application. It tackles down the *GUI mixing* challenge in our context. This is the first part of our hybrid architecture. The second part enables the communication between the pages and the controller.

6.3.2 Enabling Communication between Angular and GWT

In the hybrid architecture, all communications between pages go through the controller (see Section 6.2.1). For example, it is the case when one page displays information selected in another one. To do so, the pages need to communicate with the controller in charge of the data transmission.

In our context, a controller already exists in the source application, *i.e.* GWT application. We decided to use this controller in our hybrid architecture. Without

this preexisting controller, one would have to develop a new one. So, we need communication between the GWT pages and the GWT controller and between the Angular pages and the GWT controller. GWT to GWT communication is already supported. Because the controller is the one of GWT, and the pages do not communicate directly between themselves, Angular to Angular communication never happens.

We tackled two challenges to enable communication between Angular and GWT: calling methods, which tackles the *communication* challenge, and sending/receiving data with the *type matching* challenge.

Calling methods. The first challenge consists of allowing Angular to call methods of the GWT controller. For example, Angular calls a method of the controller to perform the navigation to another page or access logged user information. Note that, again, we only need communication between the pages and the controller since pages are independent of each other (see Section 6.2.1). To enable method invocation, we studied the Angular and GWT compilers and observed that they translate, respectively, the Java code of GWT components and the TypeScript code of Angular components to JavaScript code to be executed on the client-side (see Section 6.2.2). Thus, both are executed in the same programming language (*i.e.*, JavaScript), in the same runtime (*i.e.*, the web browser runtime). So, the Angular/JavaScript code has direct access to the methods of GWT/JavaScript.

However, we had to perform additional work to enable communication. Indeed, when GWT is compiled into JavaScript code, the Java types and methods are not exposed externally. Thus, they can not be used by another program. To expose Java methods and Java classes externally, GWT developers created the JSInterop library[3]. This library allows developers to add annotations in their code that will configure the GWT compiler. For instance, annotations will modify the generated JavaScript code to expose it to others programming languages. Thus, we rely on the JSInterop library to expose the methods of the GWT controller that needs to be called by Angular code.

Listing 6.1 presents the exposed methods with JSInterop and not-exposed methods of the GWT/Java class PhaseManager in our implementation. In this example, the class has three methods: two are exposed methods (displayPhase and addDataRefresh); and one is not-exposed (addDataRefreshEvent). The @JsMethod(<Name>) annotation (lines 2 and 7) allows one to expose methods with a programmer defined new name in JavaScript. The not exposed method, addDataRefreshEvent, is also exported in JavaScript but with an obfuscated name making it impossible to be called from Angular or any other JavaScript code. Using the JSInterop library, we ensure the *calling method* capability and tackle the *communication* challenge.

[3] http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJsInterop.html

```
1 public class PhaseManager {
2   @JsMethod(name="displayPhase")
3   public void displayPhase(PhaseMetadata pm) {
4     // ...
5   }
6
7   @JsMethod(name="addDataRefresh")
8   public void addDataRefresh(String data){
9     // ...
10  }
11
12  public void addDataRefreshEvent(String data) {
13    // ...
14  }
15 }
```

Listing 6.1: Exposed (displayPhase, addDataRefresh) and not exposed (addDataRefreshEvent) methods to hybrid architecture

In the hybrid architecture for the GWT to Angular migration, we exposed 14 methods of the controller: 9 for the navigation (including several options), 2 to check logged user rights, 1 to retrieve the currently opened page, and 2 to send and receive data during navigation and deal with the *type matching* challenge.

Sending/Receiving data. When navigating to another page, the communication can include data. We implemented a specific process when it comes to send and receive data. There are two cases: *classic communication* when data are sent and received by pages defined with the same GUI framework, *i.e.* GWT to GWT and Angular to Angular communication; and *hybrid communication* when data are sent and received by pages defined with different GUI frameworks, *i.e.* GWT to Angular and Angular to GWT communication.

For *classic communication*, a page (1) calls the navigation method of the controller with the data. Then, the controller (2) stores the data and opens the navigated page. The navigated page (3) asks the data to the controller, which, in turn, (4) returns the stored data.

For *hybrid communication*, the process is more complex. Indeed, hybrid communication suffers from the *type matching* challenge (see Section 2.3.1). So an extra step is needed to transmit data. In short, type matching is achieved through serialization and deserialization of the data. Because of its central position, this step is implemented in the controller (see Section 6.2.1).

To present the data transformation, we detail how the navigation between two

pages is done with data coming from one page, going through the controller, and being retrieved by another page.

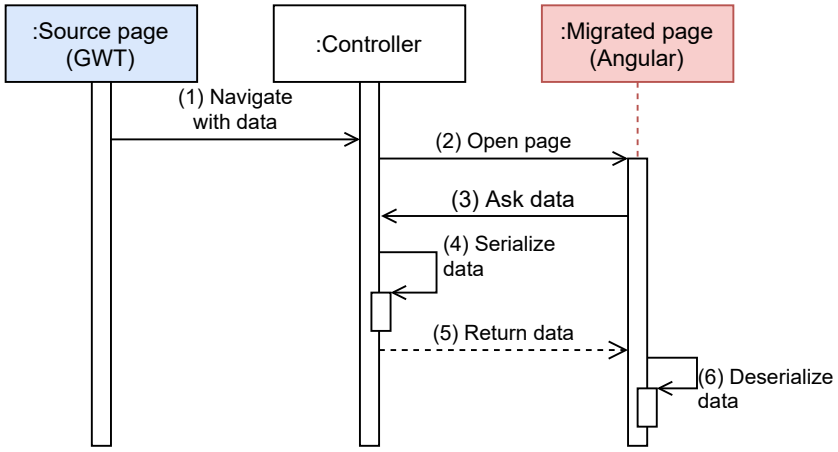


Figure 6.5: Data transformation process in hybrid communication

Figure 6.5 presents the data transformation process when navigating from a GWT page to an Angular one. First, the source page (1) calls the navigation method of the controller with the data as a parameter. The controller (2) stores the data as-is (*i.e.*, does not perform any transformation) and performs the navigation. Then, the navigated page (3) requests the data from the controller. There are two exposed methods in the controller to retrieve the data; one is used by Angular, the other one by GWT. Thus, the controller knows which framework, GWT or Angular, is retrieving the data. The controller also knows which framework sent the data by analyzing the data structure. In our context, the controller recognizes Java instances coming from GWT, and plain JavaScript objects coming from Angular. In hybrid communication, the controller (4) serializes the data in JSON and (5) returns it to the navigated page. Then, the navigated page (6) deserializes the data.

Using this serialization/deserialization strategy, we dealt with the *type matching* challenge.

We presented the implementation of our hybrid architecture to mix GWT with Angular. It deals with the *GUI mixing* challenge using Web Components, with the *communication* challenge thanks to the JSInterop library, and with the *type matching* challenge thanks to a serialization/deserialization process. Using our incremental migration approach, one can migrate applications progressively.

6.4 Conclusion

After using automatic migration tools, developers might still need to fix the migrated application manually. However, this step can take several months, and de-

velopers can not freeze all updates to end-users during this period.

To ensure the delivery of the application during the application migration process, we designed an incremental migration approach. We presented the approach in Section 6.1. It consists of incrementally migrating the application and delivering to the end-users a hybrid application containing pages developed with the source GUI framework and pages developed with the target GUI framework.

We implemented our hybrid architecture to mix GWT and Angular. It enables us to migrate applications of our industrial partner.

In the following, we validate our incremental approach and its implementation on the migration of a Berger-Levrault application.

CHAPTER 7

Incremental migration validation

Contents

7.1 Case Study: Omaje Application 109
7.2 Research Questions and Evaluation Methods 111
7.3 Evaluation Results 113
7.4 Incremental approach discussion 117
7.5 Threats to Validity 119
7.6 Incremental approach conclusion 121

We designed an incremental migration approach that enables the migration page by page of an application. It comes with a hybrid architecture that can mix various GUI frameworks into one application. In particular, we proposed an implementation to mix GWT and Angular applications.

In this section, we validate our incremental migration approach on an industrial application at Berger-Levrault. To do so, we performed the migration using our incremental migration approach (Chapter 6) and our GUI migration tool (Chapter 3).

First, in Section 7.1, we present the application to migrate. In Section 7.2, we detail the research questions and evaluation methods. In Section 7.3, we present our results. In Section 7.4, we discuss our incremental approach. In Section 7.6, we conclude our work on the incremental approach.

7.1 Case Study: Omaje Application

To evaluate our incremental approach in a real industrial set-up, we applied it to migrate a GWT application to Angular at Berger-Levrault. The application is called Omaje and was selected by its development team as representative of other Berger-Levrault GWT applications. Omaje is a client subscription management application used internally and, therefore, a safe case study for our experiment. The Omaje application includes 20 main web pages distributed into 9 modules built using 6,683 GWT graphical elements. It uses 33 kinds of widgets, from basic ones, as a button, to complex ones, as charts and tables that auto-update part of the GUI

when a row is selected. In total, in its original version, Omaje weights 191KLOC in 2,669 Java classes and 14,882 methods. The developers of Omaje roughly estimated the time to migrate the application manually to 104 person-days.

We hired (i) a Master student as a trainee to perform the migration of Visual code of Omaje, and (ii) an Angular expert engineer to migrate the Behavioral code. The Master student and the Angular expert did not know the Omaje application. The Master student had no knowledge of Angular but had experience with ReactJS (a similar GUI framework). The Angular expert engineer worked with Angular for more than three years.

Visual code migration. For the trainee, the work consisted in selecting a page, migrating it with our GUI migration tool, and fixing the visual aspect. First, he required 10 days to install the application environment, discover the Angular framework, and learn how to use our migration approach and tools. When fixing a page, the intern also encountered Custom Widgets, *i.e.*, GUI elements not migrated automatically because they only exist in the source framework (see Section 3.3.1.4). In this case, he created the corresponding widget in the target framework and added it into the widget map of our tool (see Section 4.1.1). Once the environment was installed, the trainee finalized the migration of the Visual code in 14 days. In total, the migration of the Visual code required 24 days: 10 days for the installation and 14 days for the migration.

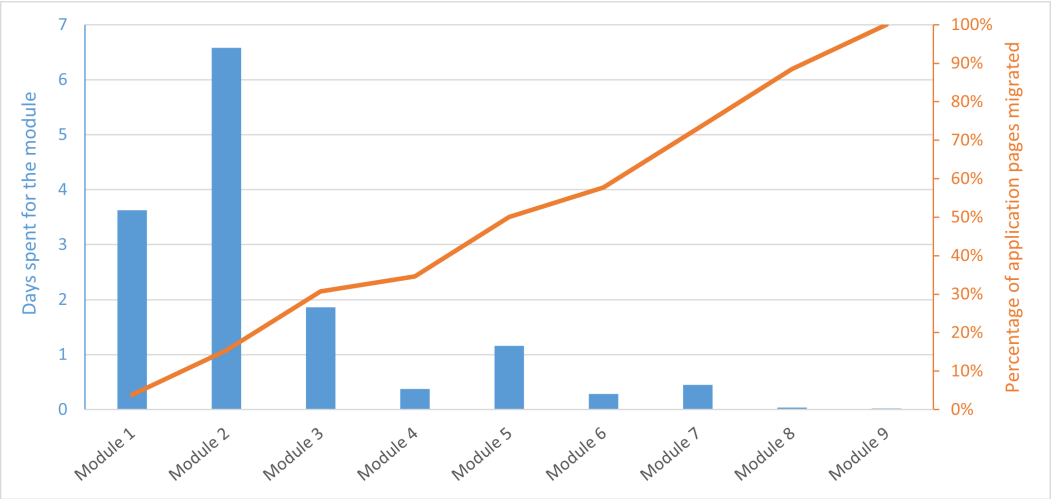


Figure 7.1: Time spent by module to migrate the Visual code

Figure 7.1 presents the amount of time in days spent by the trainee to perform the Visual code migration of each module. It required more time to migrate the first two modules. It corresponds to the time required for the intern to understand

our tool and architecture. Then, one can see that the time steadily decreases. The trainee developed all along the process Custom Widgets that were reused for the migration. Thus, by reusing newly created widgets, he spent less and less time on each module. Figure 7.1 also highlights that fixing module 2 was time consuming but do not greatly contribute to the percentage of migrated pages. For instance, the module 2 corresponds to 11% of the final application but requires 45% of the time required for the migration of the Visual code.

Behavioral code migration. For the Angular expert engineer, the work consisted in taking the GUI pages produced by the trainee, fix the Behavioral code, and integrate the migrated page into the hybrid architecture for deployment.

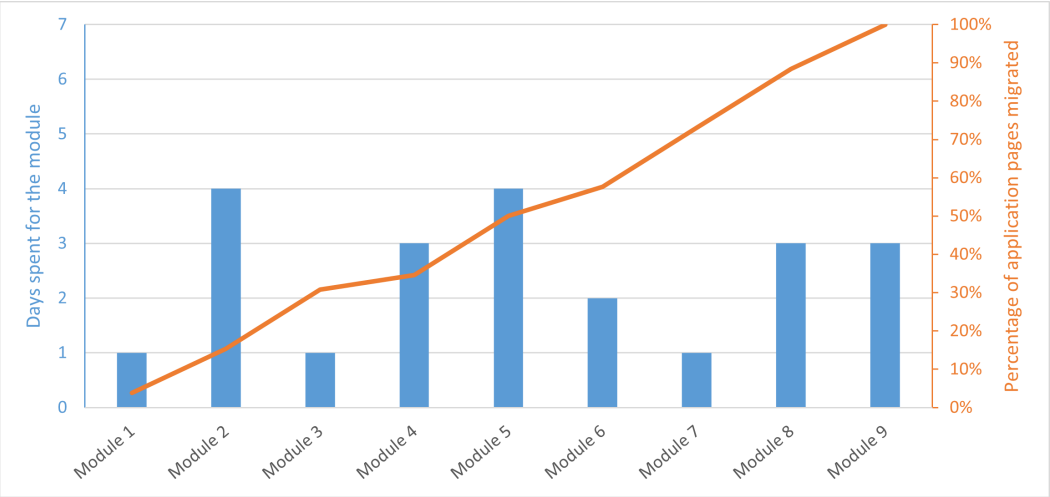


Figure 7.2: Time spent by module to migrate the Behavioral code

Figure 7.2 presents the amount of time in days spent by the engineer to perform the Behavioral code migration of each module. He required 22 days to finalize the migration. Since the engineer did not need to know our tool because he takes as input the Angular pages pre-migrated by the intern, no extra time is required at the beginning of the migration project. Contrary to the Visual code migration, the time required to migrate Behavioral code (blue bars) does not decrease with the number of remaining modules. The engineer required an average of 2.5 days and a maximum of 4 days to migrate a module.

7.2 Research Questions and Evaluation Methods

To evaluate our approach, we defined four research questions. For these RQs, we used three versions of the Omaje application: a GWT version, an Angular version,

and a hybrid version (see below).

RQ 1. Does the migrated application have the same features as the source application?

To answer this RQ, we ask the development team to execute their functional tests over the GWT and the migrated Angular applications. The application tests are detailed in a user manual provided by the development team. In total, the Omaje application has 56 functional test scenarios. Developers compare the execution results for each version of the application and report potential discrepancies in the migrated Angular one.

RQ 2. Does data communication overhead impact the speed of browsing between pages?

This RQ aims to evaluate the impact of the serialization and deserialization approach we used to tackle the *type matching* problem (see Section 6.3.2). We expect that GWT to GWT and Angular to Angular communications are fast since they do not require additional data manipulation (*i.e.*, they are handled by the controller of the hybrid architecture but do not require serialization and deserialization). Hybrid communications might require additional time but need to be imperceptible to the end-user.

To evaluate this RQ, we execute scenarios requiring communications between GWT and Angular. The scenarios include the transmission of data of different sizes. Data includes objects with attributes of different types: primitives (*i.e.*, string, int, *etc.*), collections, dictionaries, backward reference to an existing object in case of data with cycles, and other objects.

We ran the scenarios using Microsoft Edge version 87.0.664.66 on a laptop with 16 Go RAM and the Intel Core i7-7500U CPU. No other application was running on the computer during the experiment. To measure the communication time, we used the pre-built JavaScript feature `console.time()`. To avoid bias in the computed times, each scenario is run 1,000 times, and we report the average time.

RQ 3. Does the build time of the hybrid and Angular applications deteriorate compared to the GWT one?

To evaluate this RQ, we measure the required time to build the GWT, the hybrid, and the Angular application. Building an application consists of creating the `.class` files and the transpilation of TypeScript (respectively Java) to JavaScript. Build time for large applications can be time-consuming (hours), and it is essential to ensure that building the hybrid application is not prohibitively long.

RQ 4. Does the GUI performance of the hybrid application deteriorate compared to the GWT source and Angular migrated applications?

This RQ aims to compare the performance when displaying the GUI of pages between the GWT, the hybrid, and the Angular applications. We measured the exe-

cution time of 4 execution scenarios for each application. The execution scenarios are:

1. Accessing the first page of the application where initial scripts are run;
2. Accessing a middle-sized page;
3. Accessing a large page that requests and displays a lot of data;
4. Modifying data with several requests to the database and updating the page UI.

When evaluating performance for the hybrid application, the accessed pages are in Angular, whereas the controller is in GWT. Thus, for the hybrid application, this RQ evaluates if using Web Components actually tackles down the *GUI mixing* problem (see Section 6.3.1).

To measure the execution times, we used the built-in performance tool of Microsoft Edge browser[1]. It gives us detailed results on the GUI execution.We report the performance in:

- scripting: time spent executing JavaScript file;
- rendering: time spent to compute the position and visual aspect of widgets; and
- painting: time spent to display the resulting page in the web page.

We evaluated the four RQs on three applications. The GWT one is entirely written in GWT. It is the source application. The Angular one in which the pages and the controller are written in Angular. It is the migrated application. And the hybrid application with a controller written in GWT, three pages migrated to Angular and integrated into the application, and the remaining pages still in GWT. We discuss the choice of the three migrated pages of the hybrid application in Section 7.4.4.

7.3 Evaluation Results

In the following, we present the result to the research questions.

RQ 1. Does the migrated application have the same features as the source application?

[1] https://docs.microsoft.com/en-us/microsoft-edge/devtools-guide-chromium/evaluate-performance/

We gave the migrated application to the current Omaje developers and testers. They performed the 56 functional test scenarios of the application.

Developers did not report any bugs in the migrated Omaje application. From this perspective, they decided to stop supporting the old GWT application and use the Angular version for future development and deployment.

At this moment, developers have successfully developed two new features in the Angular application.

Summary: Using our approach, we migrated the application in a semantically equivalent way. The development team decided to replace the old application with the Angular migrated one.

RQ 2. Does data communication overhead impact the speed of browsing between pages?

To check the performance of the communications between GWT and Angular inside the hybrid architecture, we performed several communications between pages in GWT and Angular. During the communication, data of different sizes were exchanged, with several fields and cyclic references. We report the average time over 1,000 executions in Table 7.1.

Table 7.1: Communications performance in millisecond

Source \ Target	**GWT**	**Angular**
GWT	2 ms	49 ms
Angular	7 ms	2 ms

For the communications between pages defined with the same GUI framework, both GWT-to-GWT and Angular-to-Angular communications need 2 ms. They do not require additional data manipulation.

For the communications between pages defined with different frameworks, GWT-to-Angular communications require 49 ms, and Angular-to-GWT communications require 7 ms. So there is a cost for converting data. After investigation, we found that GWT-to-Angular communications are slower because the Angular deserialization library we used is less efficient than the GWT one. The poor performance of the Angular library we used for deserialization is a known issue[2] and might be solved in the future.

Summary: Hybrid communications cost does not much impact the end-user. Although GWT-to-Angular communication is slower, it remains imperceptible for the end-user

[2] https://github.com/pichillilorenzo/jackson-js/issues/18

RQ 3. Does the build time of the hybrid and Angular applications deteriorate compared to the GWT one?

We measured the compilation time to build each application. Table 7.2 summarizes the time required to build the GWT, the hybrid, and the Angular applications. In GWT, there are two compilations: the build of the Java project and the Java to JavaScript transpilation when first accessing the application. Building the Java project costs 366 seconds, and the transpilation requires 131 seconds. Thus, the complete GWT compilation costs 497 seconds.

Table 7.2: Building performance in second

Application	Building time
GWT	497 s
Hybrid	526 s
Angular	96 s

For the hybrid application, the build time is the same as for GWT (366 seconds). However, the transpilation time requires 160 seconds. The additional time for the transpilation comes from the usage of the JSInterop library (see Section 6.3.2). Thus, the hybrid architecture required 526 seconds.

The Angular application has a single compilation that requires 96 seconds.

Summary: This analysis shows that building the hybrid architecture is 5% slower than building the source GWT application. The Angular application build is 80% faster than the GWT one. Thus, using a hybrid architecture has no important impact on building performance. It can be used to perform incremental migration.

RQ 4. Does the GUI performance of the hybrid application deteriorate compared to the GWT source and Angular migrated applications?

We compare the performance of each application for the four scenarios detailed Section 7.2. Figure 7.3 presents the results of the performance evaluation. We perform the evaluation for home page first access, middle-size page first access, database request, and large page first access. Each bar presents the time in milliseconds reported when evaluating the GWT application, the hybrid application, and the Angular application.

For the first access to the home page, the GWT application is the fastest. GWT pre-compiles the home page during compilation, thus, home page access is fast. On the other hand, when accessing the home page of the Angular application, the Angular runtime is loaded with the application script. This step is time-consuming for the Angular application. Finally, the hybrid application is the worst case because it combines the worst of both worlds. We investigated the reasons that make

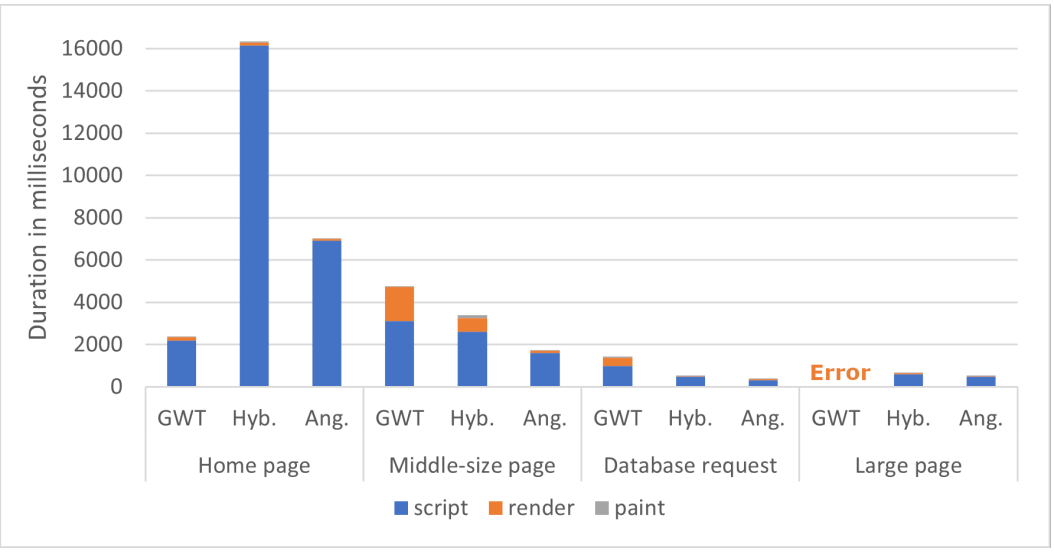


Figure 7.3: Performance evaluation result

home page access slow when using Angular, and discovered that our implementation does not use the *lazy loading* feature of Angular. Using this feature is part of our future work. Additionally, using the Ahead Of Time[3] (AOT) compilation option of Angular could improve further the performances.

For the middle-size web page access, both the hybrid and the Angular applications are faster than the pure GWT one. The hybrid application benefits from the Angular speed, and the Web Components usage does not much deteriorate the time required to render the GUI. This scenario also highlights the low performance of GWT to render the GUI as compared to Angular. This scenario shows that the hybrid and Angular applications improve the application response time for the end-user.

The third group presents the performance of the application when users request the database (for instance, to update the data). Making a request implies a scripting step where the data are serialized, sent to the back-end, new data are received, and the UI is updated. Again, the hybrid and the Angular applications are the fastest. It is due to the usage of REST/JSON instead of the GWT/RPC protocol and to the time required by GWT to *render* the new UI after retrieving the data. The usage of REST/JSON is discussed in Section 7.4.2. This scenario shows that during and after the migration, the application is more responsive to end-user interaction.

Finally, the last group presents the performance when accessing a page displaying a lot of data. In the case of GWT, the Microsoft Edge built-in plugin crashed during the evaluation. So, we do not have execution time data. This is because GWT took several minutes to process the received data and the plugin ran out of

[3] https://angular.io/guide/aot-compiler

memory. When performing the migration from GWT to Angular, the developers improved the performance of the page. In fact, they were able to use already optimized existing Angular widgets to fix this page. With the fix, the hybrid and Angular applications have better performances than the GWT one. We also note that thanks to the optimization, the large page became the fastest one. Applying optimization thanks to recent Angular widgets on all application pages could improve the overall application performance.

Summary: The hybrid application has better performances than the source one. It is also the case of the migrated Angular application. The hybrid application takes benefits of the Angular features. RQ4 validates the usage of Web Components to tackle the *GUI mixing* problem.

7.4 Incremental approach discussion

This section discusses some difficulties that we addressed to migrate GWT applications to Angular. Section 7.4.1 discusses how the industrial context might have eased the implementation of our approach and architecture. Section 7.4.2 presents the migration of the back-end services when migrating to another GUI framework. Section 7.4.3 discusses our case study and the scenarios we used to validate the incremental migration approach. Finally, Section 7.4.4 discusses how the pages migrated in the hybrid application might have impacted our result.

7.4.1 Industrial coding conventions

We perform the incremental migration approach in the context of Berger-Levrault on the Omaje application. Berger-Levrault set up several coding conventions that have eased the usage of our hybrid architecture.

In the company applications, the pages are independent one of the other. Having independent pages is a requirement of our hybrid architecture (see Section 6.2.1). So, we did not have to perform any step to make pages independent, and thus embeddable in our hybrid architecture. On the contrary, if pages were dependent on each other, one would be forced to break these dependencies before applying our incremental migration approach. The separation of pages is then similar to the creation of micro-services for the back-end where back-end features are separated one from the other [Zaragoza et al., 2021].

7.4.2 RPC vs. Rest services

When migrating web applications, we had to ensure communication between the front-end and the back-end. On the one hand, the GWT source application com-

municates with the back-end using RPC services and the GWT-RPC serialization. RPC services allow one to invoke a method of the back-end and the GWT-RPC serialization is a specific way to serialize data that is heavily tied to Java. By default, GWT uses RPC services. On the other hand, Angular uses Rest services and JSON serialization. Rest consists of accessing an external resource, and JSON is a known serialization representation.

Our incremental migration approach focused on the GUI migration aspect and therefore did not consider the switch of back-end request protocol. In our context, we had to modify the back-end communication protocol. For instance, we decided to migrate the services to Rest/JSON to use the most recent technology.

The modernization of the back-end, in particular the switch of communication protocol, is not an easy task. However, it eases the migration of the front-end. Indeed, since recent front-end frameworks are designed to use the same communication protocol as recent back-end frameworks, modernizing the communication protocol on both sides eases the migration process.

We also acknowledge that switching to Rest/JSON might have improved the performance of our Angular application as depicted in RQ4 in Section 7.3. The renovation of the back-end, including the communication protocol switch, is further discussed in our future work (see Section 8.3).

7.4.3 Case study and validation scenarios

We validated our incremental migration approach on one application and using four scenarios. Thus, our results might differ for other applications or other migration projects, *i.e.* migration from or to other GUI frameworks.

We decided to apply our approach to an actual industrial application instead of a toy tool to reduce this problem. We also asked developers that are not part of the incremental migration research team to perform the validation.

Finally, the hybrid application used for the validation (see Section 7.2) is composed of three migrated pages of different sizes, including simple and complex widgets. It allows us to use validation scenarios on a variety of pages. Thus, by selecting different types of pages, our validation gets closer to the industrial truth.

7.4.4 Pages migrated in the hybrid application

We evaluated our approach on a hybrid application composed of three migrated pages. However, as presented in RQ4 in Section 7.3, the number of pages migrated impacts our result. For instance, each migrated page cost several seconds to load the first time.

We investigated this challenge and discovered that, in our implementation, each Web Component comes with one migrated page, the controller, and the Angular

runtime. Thus, in a hybrid application with three migrated pages, the Angular runtime is imported three times. In case of a huge hybrid application with hundred of migrated pages, the required time to load the application would explode.

We already propose, in the previous section, solutions to reduce this problem. Using the Angular *lazy loading* feature will distribute the loading time on the pages instead of only the home page. Another solution is to package all the migrated pages into only one Web Component, and not one Web Component per page. Thus, the browser will need to load only once the Angular runtime. We plan to explore these options as part of our future work.

7.5 Threats to Validity

This section discusses the validity of our case study using the validation scheme defined by Runeson and Höst [2009]. The construct validity, the internal validity, the external validity, and the reliability are presented.

Again, our validation does not correspond to a formal validation consisting of research questions and their associated hypothesis testing. We instead performed one experiment in an industrial context. We also studied the industrial state of the art and reported that some companies are developing their applications using a hybrid architecture with other technologies than Java and Angular. For instance, Yesplan[4] develops a web application using Seaside and React.JS[5].

7.5.1 Construct Validity

Construct validity indicates whether the studied measures really represent what is investigated according to the research questions. The purpose of this study is to evaluate the ability to use our hybrid architecture to migrate applications with GUIs.

For developers, we validated that the build time of the hybrid architecture is not prohibitively long. We manually recorded the time needed to compile and transpile the application, which corresponds to the time spent by the developer waiting to see the application running. The scale of time required to build the application (more than 8 minutes), ensures that our conclusion is still valid, even considering some imprecision in the time manually recorded.

For the end-users, we evaluated the application behavior and usability.

For the application behavior, developers of the application performed the functional tests of the application that validates application behavior. We acknowledge

[4] https://yesplan.be/en

[5] https://www.slideshare.net/pharoproject/yesplan-10-years-later

that the functional tests might not cover all the application requirements. However, since the development team decided to keep the migrated version for future development, we consider the application correctly migrated.

For the application usability, we validated it against four scenarios. We reported the total time needed to perform the scenarios. To prevent possible bias, we selected four scenarios with different characteristics: size of the page, kind of request made to the database, and the first page in which initialization scripts are run. Time performances were recorded automatically and averaged over 1,000 runs to ensure the reliability of the results.

7.5.2 Internal Validity

Internal validity indicates whether no other variables except the studied one impacted the result.

Our validation is one industrial experiment consisting of the migration of a closed-source application. Even if we paid extra care to the tools we used to report the performance results of our hybrid architecture, it is rather difficult to isolate variables that might have impacted our results.

7.5.3 External Validity

External validity indicates whether it is possible to generalize the findings of the study.

We are aware that our results can not be easily generalized. We validated our incremental approach and its hybrid architecture against a closed-source application, and we can not publicly share the hybrid architecture implementation. Moreover, we describe a functional implementation to mix GWT and Angular GUI, but some issues might appear when mixing GUI defined with other GUI frameworks.

However, our hybrid architecture implementation is based on Web Components and the JSInterop library of GWT that are open-source projects. Thus, future research can easily reuse these projects for other hybrid architecture in a web context.

7.5.4 Reliability

Reliability indicates whether others can replicate our results.

Since our case study is a closed-source application, one can not replicate our result in the exact same context. Moreover, we do not provide any source code of the hybrid architecture.

To increase the reliability of our results, we perform the validation using standard free-to-use tools such as Microsoft Edge and pre-built JavaScript features.

We also detailed our evaluation methods to ease future researchers reproducing the same evaluation set-up.

7.6 Incremental approach conclusion

We validated our incremental migration approach on the Omaje industrial application migration. The migration was performed by two developers that did not know our tool nor the Omaje application. They used our GUI migration tool (see Chapter 3) and the hybrid architecture of our approach (Section 6.2).

The Omaje application is now migrated, and the Angular version is used by the development team. It is thus a successful migration.

During the migration, we also validated that the application is usable by end-users. To do so, we ran performance tests on the GWT application, the hybrid application (the application delivered to end-users during the migration process), and the migrated Angular application. The hybrid application and the Angular one have better performances than the GWT one. It validates that the performance of the application during the incremental migration is not an issue for end-users.

Based on the result of this evaluation, the company decided to reuse this work for the migration of the Human Resources application, *i.e.*, the biggest application of the company to migrate.

CHAPTER 8

Conclusion

Contents

8.1 Summary 123

8.2 Contributions 124

8.3 Future Work 125

8.1 Summary

Companies, such as Berger-Levrault, developed applications with a Graphical User Interface. To ease the creation of such GUI, they used GUI frameworks defining reusable widgets. However, as with any software, GUI frameworks are getting old, and companies must switch to more recent ones. Moving from one GUI framework to another is not straightforward and raised two scientific challenges: *support GUI frameworks agnostic migration* and *enable incremental migration*.

Chapter 2 reviews the scientific litterature. In particular, we review the existing GUI representations, the GUI migration approaches, and the constraints when migrating incrementally an application. We concluded that using a meta-model to represent the GUI is a common approach. However, a generic meta-model was lacking to represent GUI coming from both desktop and web environments. We also concluded that no approach exists to migrate application GUIs incrementally, ensuring the ability to mix GUI and communication between the two mixed GUI frameworks.

In Part I, we answer the first challenge: *support GUI frameworks agnostic migration*. It consists of representing GUI defined with different GUI frameworks and enabling the automatic migration of GUI defined with one GUI framework to another GUI framework.

Chapter 3 presents our approach to migrate application GUIs. It includes a separation of the GUI into the Visual code, the Behavioral code, and the Business code. We presented our migration approach and its pivot meta-model. Then, we detailed the extraction and generation of GUI using several steps and sub-steps. For

each sub-step, we provided usage examples for GUIs defined using programming languages or markup languages.

Chapter 4 details implementations of our approach for the Visual code and Behavioral code migration. We presented the extraction of GUI defined with the GWT, GXT, and Spec GUI frameworks and the generation of GUI defined with Spec2, Angular, and Seaside GUI frameworks. This chapter gives examples of concrete usage of our approach and meta-model. One can use our detailed extractors and generators to migrate applications.

Chapter 5 validates our GUI migration approach and meta-model on real migration projects. It presents metrics used for the migration validation and our results after performing the migration of closed-source and open-source projects. It reports that our approach correctly migrates the source applications. The generated applications are runnable, include most of the source widgets, and are visually equivalent to the source ones. Moreover, our generator produces Behavioral code similar to the ones that a real Angular developer might produce.

In Part II, we answer the second challenge: *enable incremental migration*. It consists of defining a migration approach that allows continuous delivery of the application during the migration process. To do so, it uses a hybrid architecture that mixes GUIs defined with different GUI frameworks.

Chapter 6 presents our incremental migration approach. It consists of an approach including a hybrid architecture allowing mixing GUI defined with different GUI frameworks. This chapter also presents an implementation of our approach and hybrid architecture to migrate GWT applications to Angular. It details the strategy used to mix GUI defined with different frameworks and to enable communication between the pages in the hybrid application. Thus, one can reproduce our work for different contexts.

Chapter 7 validates our incremental migration approach as well as our hybrid architecture. It considers the correctness of the final application (*i.e.*, fully migrated) as well as the performance of the intermediate application (*i.e.*, the application using the hybrid architecture). Finally, it shows that our incremental approach enables the migration of large applications.

8.2 Contributions

The main contribution of this thesis is an incremental GUI migration approach. It includes:

- A detailed approach to migrate application GUI;
- A meta-model representing the GUIs;
- A hybrid architecture enabling incremental migration.

8.3 Future Work

In this section, we present open issues not addressed in the thesis. These open issues provide opportunities to continue our research concerning the migration of application GUIs.

Migrating binary. In this thesis, we deal with the migration of GUI defined in programming languages or markup languages. However, some GUIs are defined using other formats. It is the case for applications defined with Access, Delphi, WinDev, *etc.*.

In these cases, the GUI extraction can not be performed by directly analyzing the source code files. Indeed, since the files are in binary format, one must discover another path to access GUI definition.

Bragagnolo et al. [2020a] explored the usage of the Access IDE capabilities to access the GUI. Instead of parsing the source files, they proposed to start the Access IDE of the project and to connect an extractor to the IDE API. Using the Access API, they can discover the defined GUIs and retrieve the widgets, attributes, and DOM.

Additional work should be done in that direction for the extraction of the Behavioral code and Business code. This future work is even more challenging because, in such applications, Visual code, Behavioral code, and Business code are often mixed, which makes their identification tedious.

Supporting various layout managers. In our context, source and target GUI frameworks used a hierarchical layout manager. However, other layout managers exist (see Section 2.1.2). It is the case of the hardcoded layout manager often used in old applications.

There are two simple solutions to deal with hardcoded layout manager using our approach. One can extend our layout package (see Section 3.3.1.3) with hardcoded layout concepts. Alternatively, one can create an extractor that converts the source layout into a hierarchical layout.

Both solutions have disadvantages. For the first solution, *i.e.*, extending the layout package, the layout manager conversion problem will persist when it comes to generating the target application. One will need to create one generator for each source layout manager and target GUI framework. For the second solution, converting the layout during the extraction means creating a more complex extractor, and the layout conversion can not be reused for other extractors.

An alternative solution is a mix of both. One can extend our layout package with hardcoded layout concepts, and the extractors extract the layout using the same layout concepts used in the source framework. Then, we need to develop an additional module in our GUI migration that performs layout manager conversion.

When generating the target application, the generator takes as input the GUI pivot model. Before performing the generation, it performs a model transformation to use the target GUI framework layout manager.

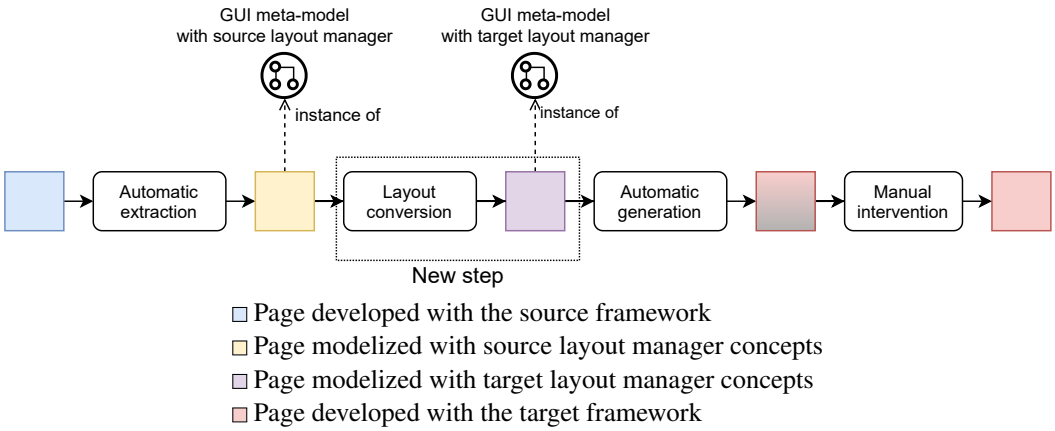


Figure 8.1: Layout manager conversion

Figure 8.1 shows the proposed future approach. Compared to our current approach, it includes a new step: "Layout conversion". This step aims to transform the layout manager concepts of the source GUI pivot model to the concepts corresponding to the target GUI frameworks layout manager. Note, the core, widget, and behavioral packages (see Section 3.2.2) are not modified during the transformation.

Back-end renovation. Our approach supports the renovation of the GUI front-end. However, when modernizing an application, the back-end is also concerned. It includes the architecture of the back-end and the front-end to back-end communication.

For the architecture of the back-end, current research focus on the back-end separation. Old applications, such as the one we migrated in this thesis, are monolithic. A monolithic application is a web application where the server-side application is a single-tier application, *i.e.* one single logic unit. Nowadays, research focuses on the separation of the monolithic back-end to micro-services [Zaragoza et al., 2021].

For the front-end to back-end communication, we already described that some modification must be performed in Section 7.4.2. It consisted of modifying the back-end in compliance with the GUI framework standard communication protocol. For instance, GWT uses Java/RPC, and Angular uses REST. Moreover, when migrating from desktop applications to web applications, there is no prior back-end, and one would have to build one using a communication protocol.

To increase the quality of the generated code, one should use a standard communication protocol. However, moving from one protocol to another is not an easy task. Indeed, one should move to another transport method, *i.e.*, in REST HTTP: GET, POST, PUT, PATCH and DELETE, and change the architectural style, *i.e.*, methods to access a resource instead of calling a method.

To switch of transport method, one can base the migration on currently existing method names. For instance, RPC methods at Berger-Levrault are prefixed with the action performed. To create data the `saveXX` prefix is used, whereas to access a data the `getXX` prefix is used. One simple transformation rule would be migrating the `saveXX` method to use the POST transport method and migrate the `getXX` method to use the GET method.

For the architectural style migration, one can base its migration rules on the original return type of the RPC methods. Easy cases are GET methods that return a data object. These methods are already getting resource-oriented. However, one still has to modify the URL path to access the resource. One idea would be to use the name of the data object and the method parameters to build the new URL.

Bibliography

Karan Aggarwal, Mohammad Salameh, and Abram Hindle. Using machine translation for converting python 2 to python 3 code. Technical report, PeerJ PrePrints, 2015. 15

Simon Allier, Salah Sadou, Houari Sahraoui, and Régis Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 214–223. IEEE, 2011. 2

Nawel Amokrane, Jannik Laval, Philippe Lanco, Mustapha Derras, and Nejib Moala. Analysis of data exchanges, towards a tooled approach for data interoperability assessment. In *Intelligent Systems: Theory, Research and Innovation in Applications*, pages 345–363. Springer, 2020. 2

Nicolas Anquetil, Anne Etien, Mahugnon Honoré Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatija Djareddir, Jèrome Sudich, and Mustapha Derras. Modular moose: A new generation of software reengineering platform. In *International Conference on Software and Systems Reuse (ICSR'20)*, number 12541 in LNCS, December 2020.

Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *arXiv preprint arXiv:1705.07962*, 2017. 13

Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse, Seriai Abderrahmane, and Mustapha Derras. Analysing microsoft access projects: Building a model in a partially observable domain. In *International Conference on Software and Systems Reuse (ICSR'20)*, number 12541 in LNCS, December 2020a. 16, 92, 125

Santiago Bragagnolo, Benoît Verhaeghe, Abderrahmane Seriai, Mustapha Derras, and Anne Etien. Challenges for layout validation: Lessons learned. In *International Conference on the Quality of Information and Communications Technology, QUATIC'2020*, September 2020b. URL https://hal.inria.fr/hal-02914750. 79, 88

Marco Brambilla and Piero Fraternali. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, 2014. 10, 12

John Brant, Don Roberts, Bill Plendl, and Jeff Prince. Extreme maintenance: Transforming Delphi into C#. In *ICSM'10*, 2010. 14, 15

John Brant, Jason Lecerf, Thierry Goubier, Stéphane Ducasse, and Andrew P. Black. Smacc: a compiler-compiler, 2017. URL http://books.pharo.org/booklet-Smacc/. 68

Jiuxin Cao, Bo Mao, and Junzhou Luo. A segmentation method for web page analysis using shrinking and dividing. *International Journal of Parallel, Emergent and Distributed Systems*, 25(2):93–104, 2010. 79, 88

Eric Cariou, Olivier Le Goaer, Léa Brunschwig, and Franck Barbier. A generic solution for weaving business code into executable models. In Regina Hebig and Thorsten Berger, editors, *21st International Conference on Model Driven Engineering Languages and Systems*, volume 2245 of *CEUR Workshop Proceedings*, pages 251–256. CEUR-WS.org, 2018. URL http://ceur-ws.org/Vol-2245/exe_paper_2.pdf. 31

Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 665–676, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180240. URL https://doi.org/10.1145/3180155.3180240. 13

David Chisnall. The challenge of cross-language interoperability. *Communications of the ACM*, 56(12):50–56, 2013. 18, 19

Santiago Comella-Dorda, Kurt Wallnau, Robert C Seacord, and John Robert. A survey of legacy system modernization approaches. Technical report, Carnegie-Mellon univ pittsburgh pa Software engineering inst, 2000. 19, 20

Gaëtan Deltombe, Olivier Le Goaer, and Franck Barbier. Bridging KDM and ASTM for model-driven software modernization. In *Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering (SEKE'2012)*, pages 517–524. Knowledge Systems Institute Graduate School, 2012. 18

Serge Demeyer, Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, and Stéphane Ducasse. Evaluating the efficiency of continuous testing during test-driven development. In *Proceedings VST 2018 (2nd IEEE International Workshop on Validation, Analysis and Evolution of Software Tests)*, pages 1 – 5, March 2018. URL https://hal.inria.fr/hal-01717343.

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. 34

Clement Dutriez, Benoît Verhaeghe, and Mustapha Derras. Switching of GUI framework: the case from Spec to Spec 2. In *Proceedings of the 14th Edition of the International Workshop on Smalltalk Technologies*, Cologne, Germany, August 2019. URL https://hal.archives-ouvertes.fr/hal-02297858.

Johan Fabry and Stéphane Ducasse. *The Spec UI Framework*. Square Bracket Associates, 2017. URL http://books.pharo.org. 55

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. ISBN 978-3-540-75208-0 978-3-540-75209-7. doi: 10.1007/978-3-540-75209-7_33. URL http://link.springer.com/10.1007/978-3-540-75209-7_33. 2, 3, 9, 11, 12, 14, 16, 17

Sergio Flores-Ruiz, Ricardo Perez-Castillo, Christoph Domann, and Simona Puica. Mainframe migration based on screen scraping. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 675–684. IEEE, 2018. 19, 20

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. doi: https://doi.org/10.1016/j.csi.2017.10.004. 9, 12, 14, 16

John Gerdes Jr. User interface migration of microsoft windows applications. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(3):171–187, 2009. 39

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. ISBN 978-989-758-194-6. doi: 10.5220/0005986002100219. URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005986002100219. 11, 12, 37

Kristian Fjeld Hasselknippe and Jingyue Li. A novel tool for automatic gui layout testing. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 695–700, dec 2017. doi: 10.1109/APSEC.2017.87. 13

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. 9, 14, 16, 17, 28, 78, 90

Mona Erfani Joorabchi and Ali Mesbah. Reverse engineering iOS mobile applications. In *2012 19th Working Conference on Reverse Engineering*, pages 177–186. IEEE, 2012. ISBN 978-0-7695-4891-3 978-1-4673-4536-1. doi: 10.1109/WCRE.2012.27. URL http://ieeexplore.ieee.org/document/6385113/. 3, 12, 25, 78, 90

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. ISBN: 0-8186-89-67-6. 17

Kostas Kontogiannis, Johannes Martin, Kenny Wong, Richard Gregory, Hausi Müller, and John Mylopoulos. Code migration through transformations: An experience report. In *CASCON First Decade High Impact Papers*, pages 201–213. Unknown, 2010. 19, 20

Christof Lutteroth, Robert Strandh, and Gerald Weber. Domain specific high-level constraints for user interface layout. *Constraints*, 13(3):307–342, 2008. URL https://hal.archives-ouvertes.fr/hal-00345425. 13

Andrew J Malton. The software migration barbell. In *ASERC Workshop on Software Architecture*. Citeseer, 2001. 14

Johannes Martin and Hausi A Muller. C to java migration experiences. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 143–153. IEEE, 2002. 15

Santiago Meliá, Jaime Gómez, Sandy Pérez, and Oscar Díaz. A model-driven development for gwt-based rich internet applications with ooh4ria. In *2008 Eighth international conference on Web engineering*, pages 13–23. IEEE, 2008. 13

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. ISBN 978-0-7695-2027-8. doi: 10.1109/WCRE.2003.1287256. URL http://ieeexplore.ieee.org/document/1287256/. 11, 12, 17, 18

Ali Mesbah and Arie van Deursen. Migrating multi-page web applications to single-page ajax interfaces. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, CSMR '07, pages 181–190, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2802-3. doi: 10.1109/CSMR.2007.33. URL http://dx.doi.org/10.1109/CSMR.2007.33. 14, 16, 18

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. ISSN 15591131. doi: 10.1145/2109205.2109208. URL http://dl.acm.org/citation.cfm?doid=2109205.2109208. 11, 12

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. ISBN 978-0-8186-6330-7. doi: 10.1109/ICSM.1994.336788. URL http://ieeexplore.ieee.org/document/336788/. 87

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. URL http://arxiv.org/abs/1807.09440. arXiv: 1807.09440. 13, 79, 88

Guillermo Polito, Stéphane Ducasse, Pablo Tesone, and Ted Brunzie. Unified ffi - calling foreign functions from pharo, 2020. URL http://books.pharo.org/booklet-uffi/. 101

Martin P Robillard and Kaylee Kutschera. Lessons learned while migrating from swing to javafx. *IEEE Software*, 37(3):78–85, 2019. 2, 3, 16, 17, 19, 20, 101

Roberto Rodríguez-Echeverría, José María Conejero, Pedro J Clemente, Juan C Preciado, and Fernando Sánchez-Figueroa. Modernization of legacy web applications into rich internet applications. In *International Conference on Web Engineering*, pages 236–250. Springer, 2011. 9, 12, 18, 25, 27

Per Runeson and Martin Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical software engineering*, 14 (2):131–164, 2009. 90, 119

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. 3, 9, 12, 16, 17

Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering*, 21(2):147–186, 2014. ISSN 0928-8910, 1573-7535. doi: 10.1007/s10515-013-0130-2. URL http://link.springer.com/10.1007/s10515-013-0130-2. 2, 9, 13, 14, 16, 39, 78, 88, 90

Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, and Jean Vanderdonckt. A layout inference algorithm for graphical user interfaces. *Information and Software Technology*, 70:155–175, 2016. 9, 12, 13, 17, 25, 27, 37, 39

Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Rahina Oumarou Mahamane, Pascal Zaragoza, and Christophe Dony. From monolithic architecture style to microservice one based on a semi-automatic approach. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 157–168. IEEE, 2020. 2

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. 3, 12, 39

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. ISBN 978-1-4503-0083-4. doi: 10.1145/1822018.1822045. URL http://portal.acm.org/citation.cfm?doid=1822018.1822045. 18, 61

Harry M Sneed and Chris Verhoef. Cost-driven software migration: An experience report. *Journal of Software: Evolution and Process*, page e2236, 2020a. 14

Harry M Sneed and Chris Verhoef. From cobol to business rules - extracting business rules from legacy code. In *Integrating Research and Practice in Software Engineering*, pages 187–208. Springer, 2020b. 31

Harry M Sneed et al. Wrapping legacy software for reuse in a soa. In *Multikonferenz Wirtschaftsinformatik*, volume 2, pages 345–360. Citeseer, 2006. 20

Werner Teppe. The arno project: Challenges and experiences in a large-scale industrial software migration project. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 149–158. IEEE, 2009. 3, 20

A. A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, November 2000. ISSN 0740-7459. doi: 10.1109/52.895180. 18, 19

Thomas Charles Terwilliger, Nicholas Sauter, and Paul D Adams. Automatic Fortran to C++ conversion with FABLE. *Source Code for Biology and Medicine*, 7 (5), May 2012. doi: 10.1186/1751-0473-7-5. URL https://scfbm.biomedcentral.com/articles/10.1186/1751-0473-7-5. 15

Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. Automatic discovery of function mappings between similar libraries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 192–201. IEEE, 2013. 15

Thiago Tonelli et al. Swing to swt and back: Patterns for api migration by wrapping. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010. 14

Marco Trudel, Carlo A Furia, Martin Nordio, Bertrand Meyer, and Manuel Oriol. C to oo translation: Beyond the easy stuff. In *2012 19th Working Conference on Reverse Engineering*, pages 19–28. IEEE, 2012. 15

Benoît Verhaeghe, Nicolas Anquetil, Stéphane Ducasse, and Vincent Blondeau. Usage of tests in an open-source community. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, IWST '17, pages 4:1–4:9, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5554-4. doi: 10.1145/3139903.3139909.

Benoît Verhaeghe, Nicolas Anquetil, Stéphane Ducasse, Abderrahmane Seriai, Laurent Deruelle, and Mustapha Derras. Migrating GWT to Angular 6 using MDE. In *12th Seminar on Advanced Techniques & Tools for Software Evolution*, Bolzano, Italy, July 2019a. URL https://hal.inria.fr/hal-02304301.

Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Abderrahmane Seriai, Laurent Deruelle, Stéphane Ducasse, and Mustapha Derras. GUI migration using MDE from GWT to Angular 6: An industrial case. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, pages 579–583, Hangzhou, China, 2019b. doi: 10.1109/SANER.2019.8667989. URL https://hal.inria.fr/hal-02019015.

Benoît Verhaeghe, Anne Etien, Stéphane Ducasse, Abderrahmane Seriai, Laurent Deruelle, and Mustapha Derras. Migration de GWT vers Angular 6 en utilisant l'IDM. In *Conférence en Ingénierie du Logiciel*, Toulouse, France, June 2019c. URL https://hal.inria.fr/hal-02304296.

Benoît Verhaeghe, Christopher Fuhrman, Latifa Guerrouj, Nicolas Anquetil, and Stéphane Ducasse. Empirical study of programming to an interface. In *Proceedings of 34th Conference on Automated Software Engineering (ASE'19)*, San Diego, United States, November 2019d. doi: 10.1109/ASE.2019.00083. URL https://hal.inria.fr/hal-02353681.

Benoît Verhaeghe, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, Abderrahmane Seriai, and Mustapha Derras. GUI visual aspect migration: a framework agnostic solution. *Automated Software Engineering*, 28(2):6, 2021a. ISSN 0928-8910. doi: 10.1007/s10515-021-00284-z.

Benoît Verhaeghe, Nicolas Anquetil, Anne Etien, Abderrahmane Seriai, Anas Shatnawi, Stéphane Ducasse, and Mustapha Derras. Migrating GUI behavior: from

GWT to Angular. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'21)*, Luxembourg City, Luxembourg, September 2021b.

Benoît Verhaeghe, Anas Shatnawi, Abderrahmane Seriai, Anne Etien, Nicolas Anquetil, Mustapha Derras, and Stéphane Ducasse. From GWT to Angular: An experiment report on migrating a legacy web application. *IEEE Software*, 2021c.

Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Abderrahmane Seriai, Anas Shatnawi, Mustapha Derras, and Stéphane Ducasse. An hybrid architecture for the incremental migration of web front-end. In *International Conference on Advanced Information Systems Engineering (CAiSE'22)*, Leuven, Belgium, June 2022. (in submission).

Leszek Włodarski, Boris Pereira, Ivan Povazan, Johan Fabry, and Vadim Zaytsev. Qualify first! a large scale modernisation report. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 569–573. IEEE, 2019. 3, 68, 89

Pascal Zaragoza, Abdelhak-Djamel Seriai, Abderrahmane Seriai, Anas Shatnawi, Hinde Bouziane, and Mustapha Derras. Refactoring monolithic object-oriented source code to materialize microservice-oriented architecture. In *ICSOFT*, 2021. 117, 126

Clemens Zeidler, Johannes Müller, Christof Lutteroth, and Gerald Weber. Comparing the usability of grid-bag and constraint-based layouts. In *Proceedings of the 24th Australian Computer-Human Interaction Conference*, pages 674–682. ACM, 2012. 13

Bo Zhang, Liang Bao, Rumin Zhou, Shengming Hu, and Ping Chen. A black-box strategy to migrate gui-based legacy systems to web services. In *2008 IEEE International Symposium on Service-Oriented System Engineering*, pages 25–31. IEEE, 2008. 19, 20