Switching of GUI framework: the case from Spec to Spec 2

Clement Dutriez, Benoît Verhaeghe, Mustapha Derras

Switching of GUI framework:

the case from Spec to Spec 2

Clement Dutriez
Université de Lille, CNRS, Inria,
Centrale Lille, UMR 9189 – CRIStAL
France
clement.dutriez@inria.fr

Benoît Verhaeghe
Berger-Levrault
France
benoit.verhaeghe@berger-levrault.com
Université de Lille, CNRS, Inria,
Centrale Lille, UMR 9189 – CRIStAL
France
benoit.verhaeghe@inria.fr

Mustapha Derras
Berger-Levrault
France
mustapha.derras@berger-levrault.com

Abstract

Developers used frameworks to create their User Interface. Those frameworks are developed in a specific version of a language and can be used until the retro-compatibility is broken. In such case, developers need to migrate their software systems to a new GUI framework. We proposed a three-step approach to migrate the front-end of an application. This approach includes an importer, a GUI meta-model and an exporter. We validate our experiment on 6 projects in Pharo. We are able to migrate 5 out of 6 projects that all conserve their widgets organization.

Keywords GUI, Modernization, Spec, Spec 2

1 Introduction

Programing languages evolve at a fast pace. For example, since 2017 they were 4 versions of Java, 4 versions of Angular and 2 major versions of Pharo. As many frameworks are developed on top of programming languages, the evolution of these languages may break the compatibility of frameworks based on them. This problem is known as backward compatibility problem.

Once a software system has used a legacy framework, *i.e.* a framework that can not be used directly in the current version of the programming language used for the company application, the developers need to develop solutions to run their applications in the last development environment. One solution is to migrate the applications to the last environment using another framework. In the case of a legacy Graphical User Interface (GUI) framework, the developers need to switch framework. For example, Swing, AWT and JavaFX have been removed from the last Java version and a new version of Spec is in progress.

Those evolution force companies to update their software systems regularly to avoid being stuck in old technologies. To reduce the time induced by such modernization many research papers proposed solutions to migrate the GUI part of applications [6, 9, 11, 12]. None of them proposed a solution adapted for the Spec framework.

To support the migration of Spec GUI, we present a three-step approach. It includes a GUI meta-model, a strategy to generate the model, and another to create the target GUI. To validate this approach, we developed a tool which migrates Spec applications to Spec 2. Then, we validated our approach on 6 real projects that used both simple and complex user interfaces. Our migration succeeded for 5 projects out of 6. Since Spec 2 is still in active development, we only tested our approach on projects that can be written with the basic components already implemented in Spec 2 *e.g.* input, text, list.

First, in Section 2, we review the literature on GUI meta-modeling. Section 3 describes the differences between Spec framework and the other GUI framework. Section 4 describes our migration approach. We present our implementation in Section 5. In Section 7, we present our results. Finally, in Section 8, we discuss the results obtained with our tool and future work.

2 Related Works

In the following we present meta-models found in the literature to represent GUI.

The OMG designed the Knowledge Discovery Metamodel (KDM) standard to support the evolution of software. The standard defined a meta-model to represent a piece of software at a high level of abstraction. It includes, among other things, a UI package which represents the components and behavior of a GUI.

The main entity of the UI meta-model 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, *etc.* UIField corresponds to a user interface widget, *e.g.* a form, a text field, a panel, *etc.* An UIResource might be composed of other UIResources to represent the DOM, and UIActions to represent the behavior of the user interface.

The Interaction Flow Modeling Language (IFML) is specialized in modeling applications with a GUI. The aim of

IWST'19, August 27–29, 2019, Cologne, Germany
2019.

IFML [1] is to provide tools to describe the visible parts of an application. In the IFML meta-model, the visible elements of the GUI are called ViewElements. A ViewElement can be refined as a ViewContainer or a ViewComponent. ViewContainer represents a container of other ViewContainers or ViewComponents, *e.g.* a window, an HTML Page, a pane, *etc.* The authors use this relation of containment to represent the DOM.

A ViewComponent corresponds to a widget which displays its content, *e.g.* a form, a data grid, an image gallery *etc.* It can be linked to multiple ViewComponentParts to represent the data inside the component. The data can be an input field inside a form, an image element of a gallery, *etc.*

In their studies, Gotti and Mbarki [5] and Sánchez Ramón et al. [10] used the KDM meta-models. Both authors added the Attribute entity to the meta-model.

Fleurey et al. [3] did not explicitly describe the GUI meta-model, but we extracted information from their navigation meta-model. They have, among other things, three elements in their UI meta-model that represent the main window of the application, the graphical entities and their associated events to represent the behavioral part of the application.

The UI meta-model of Garcés et al. [4] differs from the previous ones. In their representation, a widget corresponds to data found in a database directly displayed in a table. This difference is due to the context of their work. They migrated User Interface of Oracle Forms applications which are really different from modern GUI framework.

Memon et al. [7] designed a meta-model with two entities: widgets and attributes. An UI window is composed of a set of widgets that can have attributes.

Samir et al. [9] worked on the migration of Java-Swing applications to Ajax web applications. They created a meta-model to represent the UI of the original application. This meta-model is stored in a XUL (XML-based User interface Language) file and represents the widgets with their attributes and the layout. These widgets belong to a Window and can fire events when an input is performed. The input and the event correspond to the Action and the Event entities of the KDM model. The XUL format has been discontinued.

Shah and Tilevich [11] used a tree architecture to represent the UI. Each node corresponds to a component with its attributes. The root of the tree is the top-level component of the represented GUI.

Joorabchi and Mesbah [6] represented a user interface with a set of UI elements. Those elements are basic widgets such as text, button, input. Each element can be linked to multiple attributes and actions.

Mesbah et al. [8] did not present directly their meta-model for the user interface. However, they used a DOM-tree representation to analyze different web pages. The meta-model also includes an event entity for the behavioral part. They used different instances of their UI meta-model to represent the web pages of the application.

All the authors used the notion of widget that represents a visual entity of the user interface. And, most of them have an entity attribute that represents a characteristic of a widget. Finally, they used a tree representation and the root of this tree represent the main frame of the GUI.

3 Comparison of Spec and Spec 2

Spec and Spec 2 are frameworks that allow developers to write GUI for Pharo applications. Both divides the GUI definition into three steps:

- **Widgets organization.** Spec and Spec 2 frameworks define the GUI organization in one method in which is defined the layout of the application. At this step, apart from the layout, all widgets are abstracts but are defined using a symbol. The properties of the layout components, for example, the height, are also defined in this section.
- **Widgets initialization.** In this step a concrete User Interface component is created for each abstract widget defined in the previous step. It also defines the properties of the widget such as its label, its color *etc.*
- **Events declaration.** Developers should define the user interactions, *e.g.* click, hover, with the widgets in this step.

Even though the GUI definition is done in the same way for both frameworks, there is one notable difference. In Spec, the organization of the widgets is defined using only one class which can be used to define the layouts of the GUI. In Spec 2, the layouts are defined using multiple classes that are also User Interface elements.

4 Migration approach

This section presents the migration approach we designed. First, Section 4.1 describes the migration process. Section 4.2 presents our GUI meta-model.

4.1 Migration process

From the state of the art, we designed an approach for the migration.

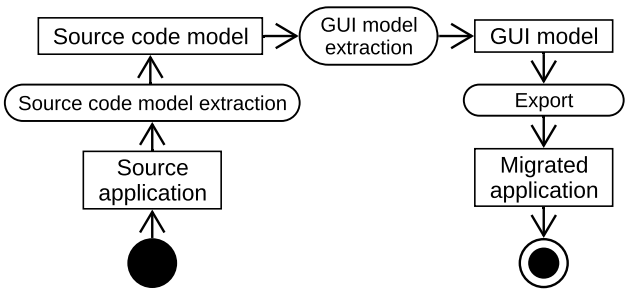


Figure 1. Our GUI migration process

Figure 1 presents the process divided into the following three steps:

1. *Source code model extraction.* We build a model representing the source code of the original application. It produces a FAMIX [2] smalltalk model which allows one to easily query the former application for the next step.
2. *GUI model extraction.* We analyze the source code model to detect the visual code elements describing the GUI. The GUI meta-model is described Section 4.2.
3. *Export.* We re-create the GUI in the target language. This step creates the classes and the necessary configuration classes to run the exported user interface.

4.2 GUI Meta-model

To represent the User Interface of an application, we designed a GUI meta-model presented Figure 2. In the following, we present the entities of the meta-model.

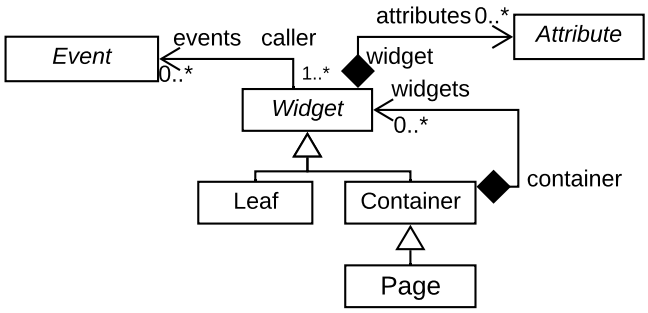


Figure 2. GUI meta-model

The main entity is Widget. It represents one component of the User Interface. It can be refined as Leaf or Container.

A container is a widget that can contain other widgets, *e.g.* a panel, a fieldset, a window. It is essential to represent the DOM heavily used in the literature. A leaf corresponds to a basic visible UI element, *e.g.* an input, a textfield.

A page is a type of container that represents a window. It corresponds to the root of an application GUI.

Finally, each widget is linked to multiple attributes and events. Attributes represent properties of the widget, *e.g.* its color, its size. Events represent all user interactions that the widget can handle, *e.g.* click, double-click, mouse hover.

5 Implementation

To test our approach, we implemented a migration tool. It is developed in Pharo using the Moose platform and allows one to migrate from Spec to Spec 2. Note that we do not migrate the behavior of the application. This will be the focus of future works.

5.1 Import

As presented Section 4.1, the extraction of the GUI model is divided into two steps: the source code model extraction and the GUI model extraction.

For the source code model extraction, we rely on a FAMIX Smalltalk importer that allows one to create a Smalltalk model for source code. Because an image does not include only the code of the old application, we must provide the list of packages in which the application is designed.

For the second step, our tool creates the GUI model form the source code model.

First, we provide to the tool the main window to extract. It is one page of the application. As presented in Section 3, a Spec UI is defined in three steps: the widgets organization; the widgets definition; and the events declaration, that will be the focus of future works.

```
1 defaultSpec
2   ↑ SpecLayout composed
3     newRow: [ :row |
4       row add: #labelA.
5       row add: #buttonA.
6     ];
7     yourself.
```

Figure 3. Widgets organization creation in Spec

For the widget organization, the tool analysis the method defaultSpec. It uses the FAMIX navigation query system to explore the DOM. Figure 3 presents an example of organization creation using Spec.

First, line 3, the tool detects the invocation of newRow:, so it creates a horizontal panel entity in the model, which is a type of container. Then, line 4 and 5, it extracts the affectations of two widgets inside the previous panel. Once the widgets are created, the tool adds them inside the panel.

```
1 initializeWidgets
2   buttonA := self newButton disable;
3     label: 'normal';
4     yourself.
5   labelA := (self instantiate: LabelPresenter)
6     label: 'normal:';
7     yourself.
```

Figure 4. Widgets definition creation in Spec

Figure 4 presents a snippet of code used to define widgets in Spec. In this user interface two widgets are created lines 2 and 5. Our tool uses the name of the attribute in which a widget is affected to link the widget organization to the widget definition. In our example, the widget found Figure 3 line 4 corresponds to the one found Figure 4 line 5.

Once the affectation of the widget is detected in widgets definition, our tool analyzes instantiate invocations to determine what type of widget should be created. To do so, it uses two heuristics: using a keyword that corresponds to a basic widgets, *e.g.* newButton, newLabel, newTextInput, or using the method instantiate: with the widget type as argument. This latter allows one to create a basic component or to create a customized one such as another user interface. Line 5, self instantiate: LabelPresenter corresponds to the creation of a text widget. Unlike the first example, the widget creation line 2, self newButton, uses the short way to create a basic widget element. So in the example, our tool creates a button widget.

Finally, our tool analyzes invocations on widgets to create their attributes. Lines 2 and 3, it creates the attributes *label*, with for the latter the value *false*, and add them to the button widget. Line 6, it creates a label attribute for the label widget.

5.2 Export

Once the GUI model is generated, it is possible to export the application. To generate the code, the tool has two visitors that respectively export the widgets organization and the widgets definition part. It also creates the minimal methods to be able to launch the application from the User Interface of Pharo.

6 Validation

In this section we present the applications we migrated with our tool to validate our approach Section 6.1. Section 6.2 presents the metrics used to evaluate our approach.

6.1 Case study

In the following, we present the 6 projects we migrated to validate our tool and approach.

- **Refactoring** is an application used by developers to refactor their code. It comes with a GUI that show the elements concerned by the refactoring and allows the developers to select which refactor actions they want to apply.
- **Setting** has a GUI to configure the settings for the Pharo code formatting configuration.
- **CriticToolbar** represents the toolbar of an application. It is composed of 3 buttons.
- **DemoButton** is part of the Demo package of Pharo. It is used to show the different configurations of a Spec button.
- **DemoForm** is also part of the Demo package of Pharo. Its GUI contains multiple type of widgets such as text, text input, check box, slider. To create the DemoForm GUI, the developers have used multiple pages with 2 pages inside the main one.
- **DBManager** provides a GUI to manage the connections between Pharo and databases. Its user interface is divided into multiple pages.

6.2 Validation metrics

We validate our approach in two steps: First, we check that all entities of interest are extracted; Second, we validate that we can re-export these widgets and that the visual aspect of the migrated application is correct, *i.e.* similar to the original one.

For the first validation, we manually count and check the entities of the user interface. Table 1 presents a summary of those numbers.

For the second validation, we check that the entities of the original applications are exported correctly. Each Page corresponds to a Pharo class that implements the necessary methods for GUI definition as presented Section 3.

We also check that the exported pages have the same visual aspect as the original ones. It is a subjective evaluation, and we are looking for an automatic solution.

7 Results and Discussion

In Section 7.1, we present the results for both the import and the export of Spec applications to Spec 2. Then, Section 7.2, we discuss the obtained results.

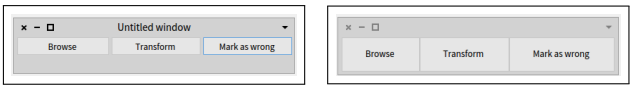
7.1 Results

Table 1 summarizes the extraction results.

For the import, our tool extracts 100% of the widgets for all the projects except the Critic one. For this latter, the widget definition does not follow the regular way to create User Interface. So, our tool cannot analyze the project.

For the Attributes extraction, our tool is less efficient. It extracts less than 25% of the attributes of the Refactoring and the Setting applications, 80% for the DemoForm and Critic projects, 74% for the DemoButton application, and 89% for the DBManager project.

For the export, we visually compared the result of the export with the original application. We present in the two following comparisons that show the main differences.



(a) CriticToolbar original **(b)** CriticToolbar exported

Figure 5. Visual comparison of CriticToolbar

Figure 5 presents the visual differences between the original (Spec) version, left hand, and the migrated (Spec 2) one, right-hand. We can see that there is only one minimal difference. The buttons do not have the same height in the exported version. This difference is due to current Spec 2 limitation

Table 1. Extraction results

	Refactoring	Setting	CriticToolbar	DemoButton	DemoForm	DBMananer
Widgets number	7	11	5	15	41 *(multiple pages)*	37 *(multiple pages)*
Widgets imported	100%	100%	*error*	100%	100%	100%
Attributes number	8	8	5	19	20	29
Attributes imported	12.5%	25%	80%	74%	80%	89%

(a) DBManager original **(b)** DBManager exported

Figure 6. Visual comparison of DBManager

Figure 6 presents the visual differences for the DBManager application. Again on the left-hand side there is the original page, on the right-hand side the page after the migration. There are many differences in this example. The text of the buttons at the left of the image is not present, the buttons at the right of the image are enabled but are disabled in the original application and are not correctly placed. Apart the last difference that comes from an application developers hack, all the differences are due to attribute extraction problems. Finally, the drop-box input has visual differences between the original and the exported application. It is due to widgets implementation. We discussed this point Section 7.2.2.

7.2 Discussion

Section 7.2.1 discuses the applications chosen for the validation. Section 7.2.2 presents how the future developments of Spec 2 can impact our solution.

7.2.1 User Interface definition

To validate our tool, we selected projects that used the Spec standard way to express widget definition. However, it is possible in Pharo to define the User Interface not extending the methods defined by the Spec framework. In this case, our tool is not able to extract the widgets of the application.

7.2.2 Widget implementations

Our tool exports the GUI with a similar aspect. This is eased because the visual aspect of basic components does not change between Spec and Spec 2. If a component is implemented differently in the target framework, the visual aspect is impacted. It is the case of the drop-box input presented Figure 6.

This lead to difficulties for automatic validation of the migrated GUI produced by our tool. It is possible to count the number of widgets and attributes but it may not be sufficient because of different visual aspect implementation.

8 Conclusion and Future works

In this paper, we exposed a preliminary work on the GUI migration problem. We proposed an approach based on a GUI meta-model and a migration process in three steps. We implemented this process in a tool to perform the migration from Spec applications to Spec 2. Then, we validated our tool on 6 Pharo projects.

We were able to extract correctly the widget organization of 5 projects out of 6. However, our tool does not extract all the attributes and works only with applications using the Spec defined methods. Dealing with all non-standard Spec application is our next challenge.

References

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

[2] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. 2011. *MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family.* Technical Report. RMod – INRIA Lille-Nord Europe. http://rmod.inria.fr/archives/reports/Duca11c-Cutter-deliverable22-MSE-FAMIX30.pdf

[3] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jezéquel. 2007. Model-Driven Engineering for Software

Migration in a Large Industrial Context. In *Model Driven Engineering Languages and Systems*, Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil (Eds.), Vol. 4735. Springer Berlin Heidelberg, Berlin, Heidelberg, 482–497. https://doi.org/10.1007/978-3-540-75209-7_33

[4] Kelly Garcés, Rubby Casallas, Camilo Álvarez, Edgar Sandoval, Alejandro Salamanca, Fredy Viera, Fabián Melo, and Juan Manuel Soto. 2017. White-box modernization of legacy applications: The oracle forms case study. *Computer Standards & Interfaces* (Oct. 2017), 110–122. https://doi.org/10.1016/j.csi.2017.10.004

[5] Zineb Gotti and Samir Mbarki. 2016. Java Swing Modernization Approach - Complete Abstract Representation based on Static and Dynamic Analysis:. In *Proceedings of the 11th International Joint Conference on Software Technologies* (2016). SCITEPRESS - Science and Technology Publications, 210–219. https://doi.org/10.5220/0005986002100219

[6] Mona Erfani Joorabchi and Ali Mesbah. 2012. Reverse Engineering iOS Mobile Applications. In *2012 19th Working Conference on Reverse Engineering* (2012-10). IEEE, 177–186. https://doi.org/10.1109/WCRE.2012.27

[7] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. GUI ripping: reverse engineering of graphical user interfaces for testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on* (2003). IEEE, 260–269. https://doi.org/10.1109/WCRE.2003.1287256

[8] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web* 6, 1 (2012), 1–30. https://doi.org/10.1145/2109205.2109208

[9] Hani Samir, Amr Kamel, and Eleni Stroulia. 2007. Swing2Script: Migration of Java-Swing applications to Ajax Web applications. In *14th Working Conference on Reverse Engineering (WCRE 2007)*.

[10] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. 2014. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering* 21, 2 (2014), 147–186. https://doi.org/10.1007/s10515-013-0130-2

[11] Eeshan Shah and Eli Tilevich. 2011. 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* (2011). ACM, 255–260.

[12] Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Abderrahmane Seriai, Laurent Deruelle, Stéphane Ducasse, and Mustapha Derras. 2019. GUI Migration using MDE from GWT to Angular 6: An Industrial Case. Hangzhou, China. https://hal.inria.fr/hal-02019015