Pyramidion: a framework for domain-specific editors

Yann Le Goff[1,2,*], Pierre Laborde[2], Alain Plantec[1] and Éric Le Pors[2]

[1]Univ. Brest, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
[2]THALES Defense Mission Systems France - Etablissement de Brest, 10 Avenue de la 1‘ere DFL, 29200 Brest, France

Abstract

Developing User Interfaces (UI) for software can be tedious without tools to visualize, modify and test. A Graphical User Interface (GUI) builder could simplify the development process of an UI. In this paper we present Pyramidion, a framework to implement domain-specific editor to show and modify the properties of any objects. We used Pyramidion to generate Pyramid, a GUI builder that can modify the properties of graphical elements that compose an UI. In this paper we will present the general concept behind Pyramidion through the example of Pyramid.

Keywords

Smalltalk, Pharo, domain-specific editor, GUI builder, UI

1. Introduction

In an industrial context, a prototype is a serious application that contains the interfaces of the final product. At *Thales Defense Mission System* (DMS), our prototypes are used during demonstration in front of end-users to benefit from their feedbacks, to experiment new ideas and finally to validate the design [1]. Real-time modifications of a prototype are often done during demonstrations to immediately take into account the suggestions of end-users and directly validate the new modification with them, improving the quality of the solution. Thus, we need tools to minimize the cost of such a change. Typically, a GUI builder can help.

In this paper we present *Pyramidion*. Pyramidion consists in a framework to implement domain-specific editor to show and modify the structure and the properties of a collection of objects. With Pyramidion new features of an editor are implemented through a plugin pattern. Such a feature can be a domain-specific editing functionality or a transversal functionality such as *undo-redo* or serialization.

We use Pyramidion to implement *Pyramid*, our GUI builder for *Pharo* [1] and the *Bloc UI library* [2]. At Thales DMS, we use Pyramid either to create new views from scratch or to change existing projects views. We also use Pyramid to change dynamically our GUI during presentations without stopping the running application.

In this paper, we present the following contributions: the definition of a domain-specific editor in Section 2, an example of a domain-specific editor in Section 3, the Pyramidion core architecture in Section 4, how we implemented Pyramidion in Pharo in Section 5, pyramid, the GUI builder created with Pyramidion in Section 6, related works in Section 7, finally we conclude the paper in Section 8.

2. Definition of a Domain Specific Editor

A domain-specific editor is an application that helps a user to modify or to create a domain-specific model composed of domain-specific objects. Improving software can be long and tedious [2] without a good understanding of its structure. An editor with a dedicated UI to modify the different objects of the

IWST 2024: International Workshop on Smalltalk Technologies, July 9–11, 2024, Lille, France

*Corresponding author.
yann.legoff1@univ-brest.fr (Y. Le Goff); pierre.laborde@fr.thalesgroup.com (P. Laborde); alain.plantec@univ-brest.fr (A. Plantec); eric.lepors@fr.thalesgroup.com (: Le Pors)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]Pharo website on: https://pharo.org/
[2]Bloc open source repository on: https://github.com/pharo-graphics/Bloc. Accessed on 21/05/2024

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

CEUR-WS.org/Vol-3893/Paper09.pdf

model can be faster than a text-based one and can help unexperimented users to better understand the domain in which they are working [3].

We developed Pyramidion, a framework to create domain-specific editors. Pyramidion reduces the effort the developers need to make to create a working editor for a specific-domain. They have to specify the accessors and mutators of the model, the associated views, and the needed editor behaviors for the specific-domain.

3. Example of a domain-specific editor

Figure 1 shows a domain-specific editor view for creating and editing a traffic light view. The traffic light view is composed of graphical elements. A graphical element is an object that can be rendered as a 2D representation on the screen. All 2D graphics libraries aim to instantiate and configure graphic elements. Considering object-oriented libraries, a graphical element is modeled as a *Composite*. Each graphic element has properties for its size, its shape, its background color, etc.

The domain-specific model of the figure 1 is composed of graphical elements and their properties. The domain-specific editor of the figure 1 displays multiple representations of the same domain-specific model. The editor is composed of 3 panels:

- On the left (1), a panel that is made of tabs, currently there is one tab named *Tree*. It shows a kind of inspector for the graphic element tree that is manipulated by the domain-specific editor; the inspector allows the user to select one or several elements of this tree;
- On the center (not numbered), a panel to render the traffic light (the domain model view) and the current selection. Any valid change in the graphic element tree is rendered on this panel;
- On the right (3), a panel also made of multiple tabs; the tab named *Visuals* is selected. The associated panel contains a specific editor for the background color property of the currently selected graphic element (2);

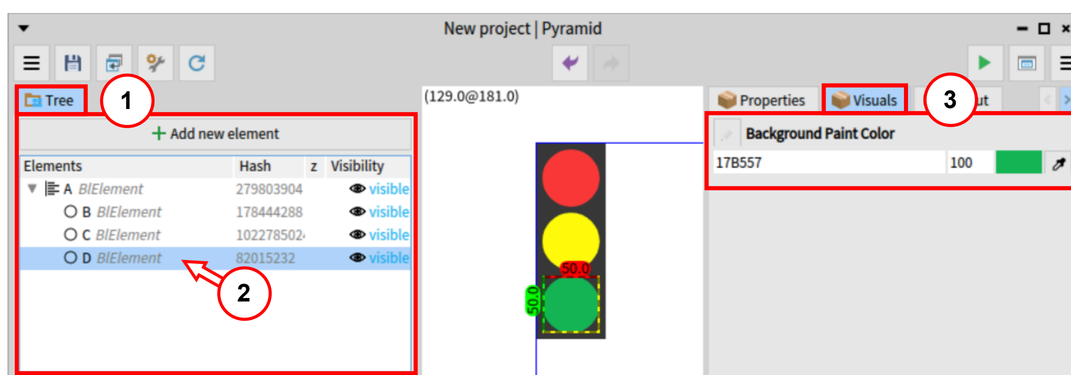

Figure 1: A Pyramidion specific editor of a traffic light view

In order to edit a particular property (e.g. the background color) of a graphical element inside a tree, a user first selects the targeted graphic element in the tree inspector. Then the property views are updated according to the selected graphic element. Finally, the user can use the specific editor for the background color property to see the current color of the selected elements and change its value. Therefore, a domain-specific editor is made of 3 subsystems:

- `Panel Builder Manager`, to modify and extend the user interface of the editor.
- `Selection System`, to select the elements inside a tree of elements.
- `Command System`, to view and update the properties of the selected elements.

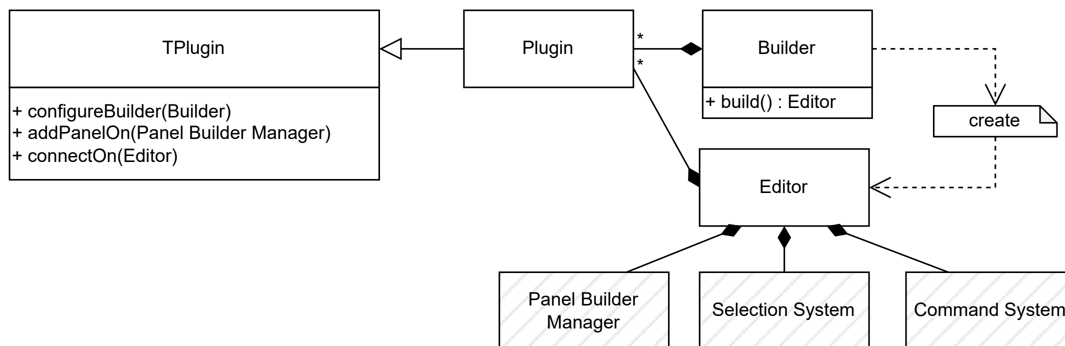

Figure 2: The class model of the architecture of Pyramidion.

4. Pyramidion architecture

The architecture of Pyramidion is based on the use of *plugins*. The architecture is shown in Figure 2. In the context of Pyramidion, a plugin is programmed as a class to add specific features to a domain-specific editor. All plugins of a specific editor are managed by a *builder*. Then, a specific editor consists in a builder and a set of plugins. Any plugin can interact with the 3 subsystems of the specific editor: the `Panel Builder Manager`, the `Selection System` and the `Command System` (in grey in Figure 2).

For the example in section 3 we need 3 plugins:

- *MainUIPlugin*, a plugin to describe the different panels of the UI of our editor.
- *TreePlugin*, a plugin to select the graphical elements.
- *BackgroundPlugin*, a plugin to display and modify the current value of the background.

Figure 3 presents a part of the build sequence of the example. The builder implements three phases for the building of a specific editor: (1) the configuration phase, (2) the view modification phase and (3) the editor building phase. Each phase is runned on all registered plugins. Regarding the traffic light example :

- During the configuration phase, `MainUIPlugin` creates a new instance of `PanelBuilderManager` (*create* (phase 1) in the Figure 3) and inject it as default value inside the builder (*modify* (phase 1) in the Figure 3).
- During the view modification phase, all plugin can modify the `PanelBuilderManager` of the builder (*extend* (phase 2) in the Figure 3). The `TreePlugin` and the `BackgroundPlugin` use the new `PanelBuilderManager` to modify the view of the editor. The `TreePlugin` adds a tree inspector view to show the graphical element tree (number 1 and 2 in Figure 1). The `BackgroundPlugin` adds a specific editor to show and modify the background of a graphical element (number 3 in Figure 1).
- During the editor building phase the builder creates the editor with the `PanelBuilderManager` (*create* (phase 3) in the Figure 3). The editor is composed of the `PanelBuilderManager` created by the `MainUIPlugin`, a `Selection System` created by the editor and a `Command System` also created by the editor. The plugins can use any of the subsystems of the editor (*extend* (phase 3) in the Figure 3). The `TreePlugin` uses the `Selection System` to update the current selection. The `BackgroundPlugin` uses the `Selection System` to access the current selection and the `Command System` to modify the graphical elements.

Each phase running consists in the sending of a specific message by the builder to its registered plugins. Therefore a plugin class must implement 3 methods: `#configureBuilder:` for the configuration phase, `#addPanelOn:` for the panel adding phase and `#connectOn:` for finalization of the editor building.

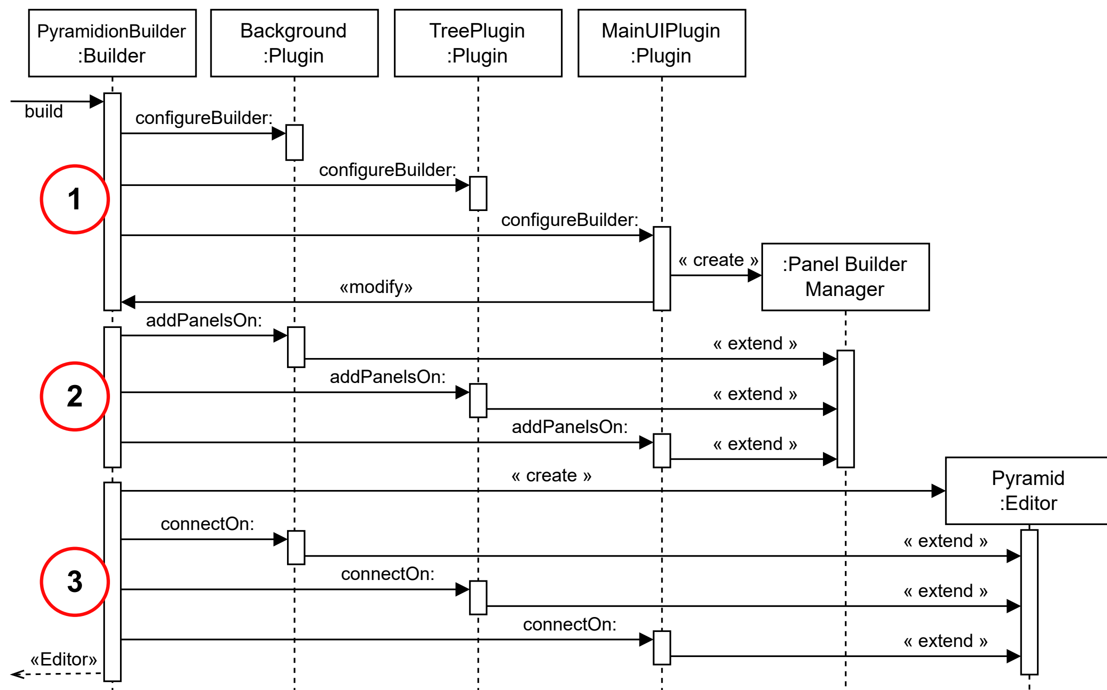

Figure 3: The build sequence of the builder and 3 associated plugins.

The next sections describe how the builder uses theses 3 plugins to construct the domain specific editor.

4.1. Configuration of the builder and creating the view of the editor

The view of a domain-specific editor is composed of *widgets*. The widgets are: the buttons, the lists, the panels, the textfields, the menus etc. The users will use the widgets to see and modify the domain-specific model or to use transversal capabilities (like the undo-redo). Pyramidion uses the widgets of the system to display the UI of the domain-specific editors. The `Widget Adaptaters` are used as *adapters* between the widgets of the system and the Pyramidion editor.

The `Widget Adaptaters` are created by the plugins using the `Panel Builders` of the `Panel Builders Manager` (see Figure 4). Then they are added to the `Panel Builder` that created them.

The `Panel Builders Manager` contains multiple `Panel Builders`. Each `Panel Builder` assembles the different widgets inside the `Widget Adaptaters` to form a *panel*. `Panel Builders Manager` uses the panels (that contains the different widgets) to create the view of the editors.

There are different types of `Panel Builder`. Each type of `Panel Builder` depends on the necessary widget types to create the domain-specific editor. If developers want to add a new widget type for its editors, they will have to create a dedicated `Panel Builder` that creates those new widgets and assembles them in a panel. For our current use of Pyramidion, 4 different `Panel Builders` are defined :

- `Notebook`, where each widget is a new view inside a tab.
- `Buttons bar`, where each widget is a button.
- `Single panel`, where only one widget corresponding to a view is displayed.
- `Menu`, where each widget is a menu item or a group of menu items.

The method `#configureBuilder:` of the plugins is used by the builder to configure itself (see phase 1 in Figure 3). Any plugin can change the default `Panel Builders Manager` of the builder, it also have to define the different `Panel Builders` that will be used to create the editor.

The `#addPanelOn:` of the plugins is used by the builder to create new `widget Adaptaters` during phase 2 in Figure 3. Any plugin can add any number of `widget Adaptaters` to the different `Panel Builders`. The `Widget Adaptater` comes with a priority value used when multiple widget Adapters want to be added on the same `Panel Builders`. The priority value will be used to define the order of the widgets on the `Panel Builders`. This priority value is an arbitrary value defined by the developer of the plugin.

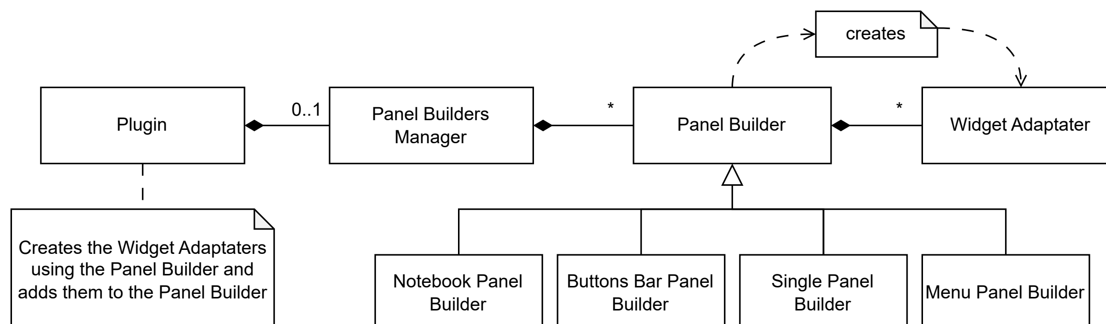

Figure 4: Architecture of the panel builders manager.

In our example The `Panel Builders Manager` is created by the *MainUIPlugin* during the configuration phase (see Figure 3). The `Panel Builders Manager` has two `Notebook Panel Builders` to add the `widget Adaptaters` created by the two other plugins.

The *TreePlugin* adds a tree inspector view to display and select the different graphical elements (see number 1 in Figure 1). The *BackgroundPlugin* adds a specific editor for colors to display and modify the background of selected graphical element (see number 3 in Figure 1).

4.2. Connect to the subsystems of the editor

The method `#connectEditor:` is used by the Pyramidion builder to link the different subsystems of the domain-specific editor to the plugins (see Figure 3).

The *TreePlugin* will have to update its content and its selection accordingly to the selection system contents and selection. It will also have to change the selection system when the user select a new element in the UI.

The *BackgroundPlugin* will have to connect to the selection system to display the correct information according to the current selection. It will have to also connect to the command executor (see Figure 5) system to be able to modify the background of the selected graphical elements.

4.2.1. The selection system

Access to all the elements of the domain-specific model is necessary to allow the users to understand the structure of the elements and the global properties of the model. Access to a specific subgroup of elements of the model is also necessary to see and modify the detail of the model. Therefore, a domain-specific editor must have 2 collections of objects:

- The first level elements, that represent all the roots of the current domain-specific model.
- The current selection, that represent a subpart of the domain-specific model.

In the Figure 1 at number 1, the element named *A* has 3 children named *B*, *C* and *D*. We can see on number 2 that the element *D* is selected. The first level elements is a list that contains the element *A* and the current selection is a list that contains the element *D*.

How plugins are dynamically linked ? The first level elements and the current selection are both observable collections. When a modification is made to one of them, a signal is sent to all the objects that observe them. Any plugin can register as an observer of the first level elements or the current selection.

This observer/observable pattern is used by the *BackgroundPlugin* to update the displayed properties panel when the current selection changes (see number 3 in Figure 1). The *TreePlugin* can modify the current selection collection and therefore trigger the signal to all the observers of the current selection.

4.2.2. Object modification mechanism

We call `property` the association of a `command` and an `input` as depicted in the Figure 5.

The `input` is a kind of specific editor that are used to display and modify the values of the selected objects. In Figure 1, the element *D* is selected. The element *D* has a green background. We can see the property for the background is present and display the green value (number 3 on the figure).

The `command` is used to retreive the value of the properties to be displayed in the `input`, mutate the selected objects (change the value of the object) and decide if the `input` should be visible to the users. For example, we do not want to display the `input` to change the text of a graphical element if no element in the selection is a text element.

Display the value of the property. When the current selection is updated, each object of the selection is tested by the verification method (`#canBeUsedOn`) of the `command` of each property:

- If the verification methods return `false` for all selected elements, then the property is not displayed.
- If only one method return `true`, then the input for this element is displayed (single selection on Figure 6).
- If multiple methods return `true`, then the property create multiple inputs (multi selection on Figure 6).

Change the value of the property. The `command` is used to mutate the selected objects when the users interact with the inputs of the properties. The `input` contains a callback that will use the `command` on the `commands executor` with the new value of the `input` as arguments for the `command` (see the method `#useCommandOn` of the `Command Executor` in the Figure 5). The `commands executor` can be extended by any plugin to add behavior before or after the execution of the command.

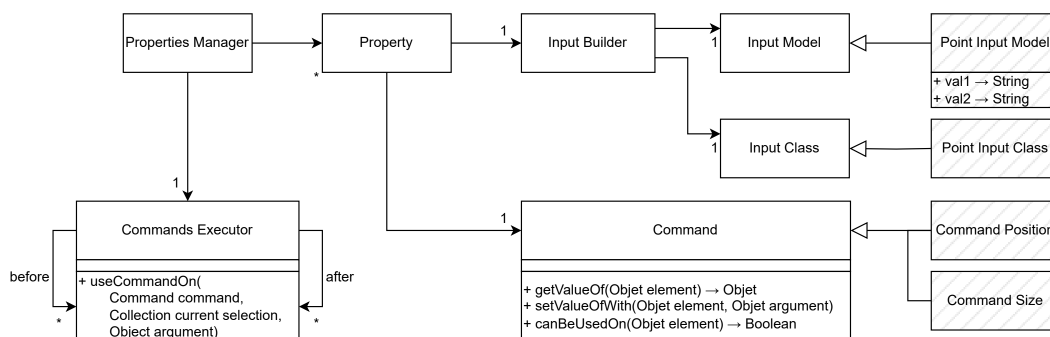

Figure 5: Architecture of the Properties Manager. The hashed part correspond to the class needed for the example in Figure 7.

Using a builder to create the inputs. The selection system allows the multi-selection of elements as depicted in Figure 6. The property uses an `input builder` that creates the different inputs when one

element is selected or in multi-selection. The `input builder` take an `input class` and an `input model` to create the inputs and their associated callbacks.

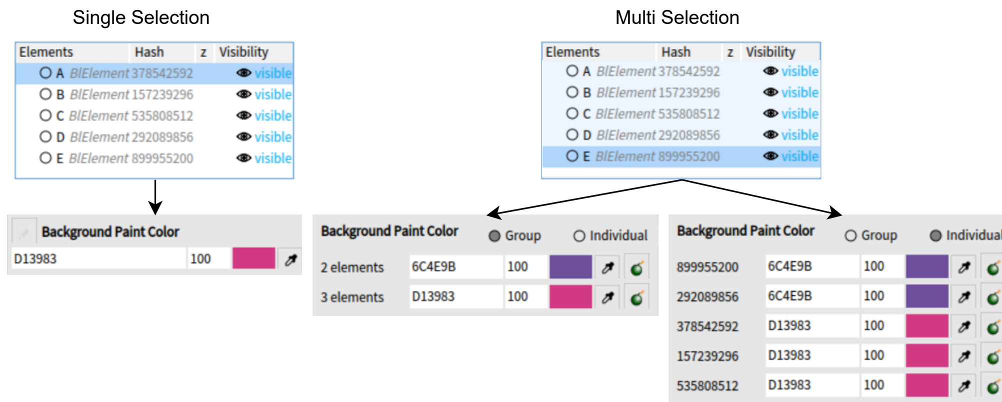

Figure 6: Example of the multi-selection on the property view.

There are different types of input class: one for the strings, one for the numbers, one for the colors, one for a boolean value, etc. Some inputs can be used in different context (like in the Figure 7). For example the position property and the size property expect a `Point` as an argument, but this point represents two different concepts. The position is a coordinate x and y on the screen, the size is a *height* and a *width*. To specialized the `Point Input` we used the associated input model (`Point Input Model`) to help the users to better understand what is the property they are interacting with.

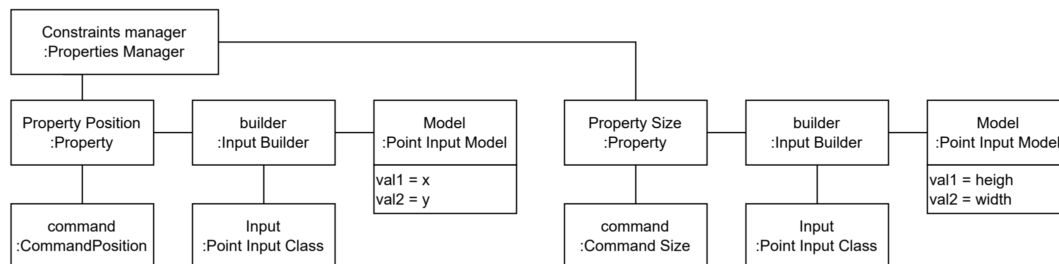

Figure 7: Implementation of 2 properties using the same input class.

5. Implementation of Pyramidion

We validate the Pyramidion architecture by implementing it in Pharo [3], a Smalltalk inspired language. Smalltalk allows the modification of the model at runtime and gives the possibility to the users to see each incremental modification. Therefore it reduces the cost of validation. Pharo is a full Integrated Development Environment (IDE) with plenty of support tools like inspectors and debuggers The editors created with Pyramidion can benefit from these tools.

The plugins in Pharo To create a plugin in our implementation of Pyramidion in Pharo, we use *Traits* [4]. The developers create a new class that implements the trait `TPyramidPlugin`. The `TPyramidPlugin` adds to the class the necessary methods for the Pyramidion builder. By default, the trait has no behavior inside the methods. If the developers want their plugins to do something, they have to override the different methods used by the builder.

[3]Pharo website on: https://pharo.org/. Accessed on 25/05/2024

A singleton to centralize all plugins To facilitate the creation of editors by the builder, we created a system to register all plugins into a singleton. The singleton centralizes all the plugins and creates a builder with all the plugins ready to be installed. Therefore, the developers only have to register their plugins to the singleton only one time, and use the builder of the singleton to create an editor. We use the initialization method of the meta-class of the plugins to register them automatically when the class is created in the Pharo image. For example, we can use this behavior to automatically add new plugins after loading the package that contains the plugins from a *Git* repository.

The Spec2 library to make the UI We chose to use the *Spec2* library [5] to make the UI of our editors. Spec2 is the default UI library of Pharo. All the widgets of the `Widget Adaptaters` and the panels created by the `Panel Builders` are created using the Spec2 library.

6. Validation

We developed a GUI builder with Pyramidion in Pharo. Some of the plugins we developed offer transversal capabilities and can be used in different domains. This is the case for:

- The serialization plugin that allows us to save the current model modifications.
- The historialization plugin keeps the state of the elements before the modification allowing the developers to undo their modifications.
- The playground plugin that creates a Pharo Playground inside the editor.

We will present Pyramid, our GUI builder for Bloc in Section 6.1. We will present a selection of the transversal plugins we created in Section 6.2.

6.1. Pyramid: a GUI builder with Pyramidion

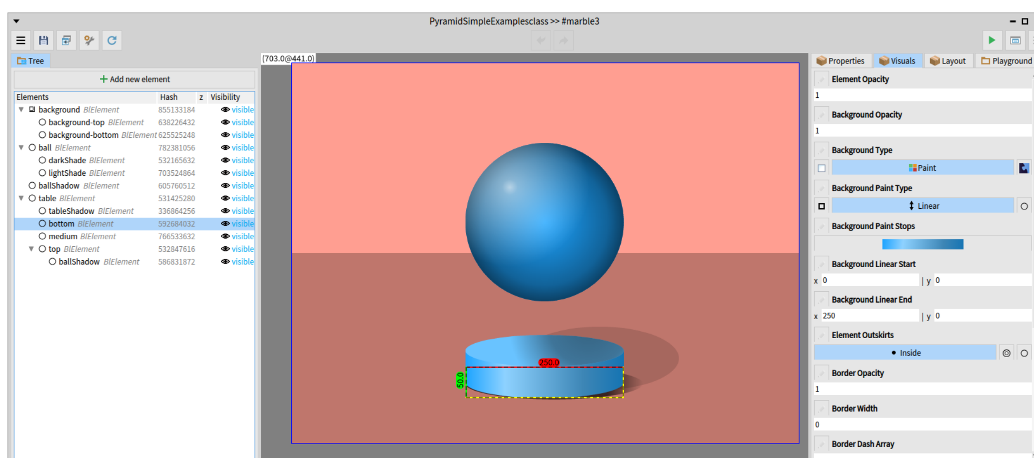

Figure 8: Example of a Pyramid editor.

We used Pyramidion to develop Pyramid [4]: a GUI builder for Pharo. Pyramid is based on Bloc, a UI library for Pharo. Pyramid allows us to modify the properties of `BlElements`, the graphical elements of Bloc. The Figure 8 shows an example of Bloc graphical elements assembled to create a 3D effect. The dynamic graphical feedback that Pyramid enables makes developing such a complex view easier than writing the script in a workspace, running it, and then observing the result multiple times.

Developers can use Bloc to develop beautiful and complex applications [1]. When we started the development of Pyramid in 2023, the source code of Bloc underwent important changes. We used the

[4]Pyramid open source repository on: https://github.com/OpenSmock/Pyramid. Accessed on 25/05/2024

plugin-based architecture to develop Pyramid with one feature corresponding to one plugin. Therefore when one of our features became obsolete because of the changes in the source code of Bloc, we only had to update a single plugin.

The default version of Pyramid is composed of 16 plugins. There are multiple plugins for the properties, there are plugins for the `WYSIWYG` (What You See Is What You Get) view, etc. At Thales DMS, we created other plugins for our specific needs. For example, we have a plugin to import designs from Figma [5] into Pyramid.

Generally, a GUI builder can only modify applications created with the GUI builder [3]. The live environment of Pharo allows us to modify the UI of any Bloc applications in real time and not only applications created with Pyramid. For example, users could be interacting with a UI developed in Bloc. While interacting with the UI, users change the current state of the application. We can use Pyramid to stop the user interactions and open the current UI inside Pyramid. Pyramid can be used to modify the UI while keeping the state of the application. When Pyramid is closed, we allow users to interact with the UI again. The UI will retain the modifications made with Pyramid.

6.2. Transversal plugins

We developed multiple transversal plugins that can be used in different domain-specific editors. Currently, the transversal plugins are used by Pyramid. We will see the serialization plugin, the historialization plugin, and the playground plugin.

The serialization plugin The serialization allows the users to save their current modifications. The serialization plugin will use the first level elements of the selection system as an entry point to generate a string that can be used to recreate similar objects. We chose to use *Ston* [6, Chapter 9] as the support for the serialization.

The historialization plugin The historialization is important because it is one of the most used commands in other IDE [7]. The historialization plugin extends the `command executor` to add behavior before and after the execution of the commands. The plugin will create a memento of the state of the objects before the execution of the command and afterward. The mementos are used to restore the state of the objects.

The playground plugin The playground allows to add a Pharo playground in any domain-specific editor. The playground of Pharo allows to send any message to any object. It is a possible entry point to connect any tools of Pharo to the domain-specific model. It can also be used to modify any properties that are not already developed in the domain-specific editor.

7. Related works

Domain-specific tools Moldable debugger [8] proposes a domain-specific framework for debugger. The moldable debugger allows developers to create domain-specific extensions to their debuggers to resolve domain-specific problems. Pyramidion can be used to understand, modify, and create a new domain-specific model. The debugging capability of Pyramidion is limited but Pyramidion can be used with the other tools available in Pharo, like the debugger. The moldable debugger can move seamlessly from one domain to another thanks to its user interface and its activation predicate for domain-specific tasks, our approach can also adapt its UI for different domain properties.

The Eclipse platform [6] allows the integration of domain-specific tools. The Eclipse platform developer chooses which tools are installed. With Pyramidion, the developer also chooses which plugins are installed. The Eclipse platform offers different levels of integration [9]. Some tools will interact with

[5] Figma website on: https://www.figma.com/. Accessed on 25/05/2024
[6] Eclipse website: https://eclipseide.org/. Accessed on 10/08/2024

the Eclipse platform through data sharing. Others will offer dedicated API. Finally, some can modify the UI of the Eclipse IDE, respond to a change in the selection, and even respond to an event created by other tools. Pyramidion has only one level of integration, corresponding to the UI integration.

GUI builders Citrus [10] is a language and a framework used to create a graphical editor for structured data and code. Pyramidion can also modify structured data but currently, it has not been tested to modify code. Citrus can be used to define models and behaviors of applications. We would like Pyramid to do the same soon. Our implementation of Pyramidion does not define a new language but uses an already specified one, the Pharo language.

Amulet [11] is a development environment to create GUI. Like Pyramidion, Amulet allows the modifications of graphical objects at runtime. Unlike Pyramid, Amulet also allows the modifications of the behaviors of the application. Amulet provides the developer debugging tools: an inspector and a debugger. In Pyramid, we created plugins to access the already present debugging tools inside the Pharo environment. In Pyramid, we have access to the Pharo inspector and the Pharo playground. From the inspector, we can set breakpoints and object-centric breakpoints [7].

8. Conclusion and Future Works

We developed Pyramidion, a plugin-based framework that allows developers to create their domain-specific editors. The plugin-based architecture simplifies the development process and permits the reuse of some of the defined behavior between really different domains. We then used Pyramidion to create Pyramid, a GUI builder in Pharo.

As discussed in the previous sections, Pyramid cannot modify the behavior of the user interface and dynamic widgets. But thanks to the Pyramidion framework, it will be easier to add new functionality to Pyramid. This would allow Pyramid to create or modify complete applications at runtime with a dedicated UI, allowing us to prototype more quickly.

References

[1] P. Laborde, É. Le Pors, Y. Le Goff, A. Plantec, S. Costiou, Ergonomic evaluations of human-machine interfaces in the defense business: an example of a collaborative maritime surveillance system, 2024.

[2] D. C. Littman, J. Pinto, S. Letovsky, E. Soloway, Mental models and software maintenance, Journal of Systems and Software 7 (1987) 341–355. URL: https://linkinghub.elsevier.com/retrieve/pii/0164121287900331. doi:10.1016/0164-1212(87)90033-1.

[3] A. Ply, Software development in ada and motif using a gui builder, volume 2, 1996, pp. 505 – 510 vol.2. doi:10.1109/NAECON.1996.517696.

[4] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. P. Black, Traits: A mechanism for fine-grained reuse, ACM Transactions on Programming Languages and Systems 28 (2006) 331–388. URL: https://dl.acm.org/doi/10.1145/1119479.1119483. doi:10.1145/1119479.1119483.

[5] J. Fabry, S. Ducasse, The Spec UI framework, Square Bracket Associates, [Kehrsatz] Switzerland, 2017. OCLC: 1021885954.

[6] D. Cassou, S. Ducasse, L. Fabresse, J. Fabry, S. Van Caekenbergh, Enterprise Pharo a Web Perspective, Square Bracket Associates, Place of publication not identified, 2016. OCLC: 1151080859.

[7] G. Murphy, M. Kersten, L. Findlater, How are Java software developers using the Eclipse IDE?, IEEE Software 23 (2006) 76–83. URL: https://ieeexplore.ieee.org/document/1657944/. doi:10.1109/MS.2006.105.

[8] A. Chiş, T. Gîrba, O. Nierstrasz, The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers, in: B. Combemale, D. J. Pearce, O. Barais, J. J. Vinju (Eds.), Software Language

[7] Object-centric debugging on Pharo: https://thepharo.dev/2020/07/16/object-centric-breakpoints-a-tutorial/. Accessed on 10/08/2024

Engineering, volume 8706, Springer International Publishing, Cham, 2014, pp. 102–121. URL: http://link.springer.com/10.1007/978-3-319-11245-9_6. doi:10.1007/978-3-319-11245-9_6, series Title: Lecture Notes in Computer Science.

[9] J. Amsden, Levels of integration: Five ways you can integrate with the eclipse platform (2001).

[10] A. J. Ko, B. A. Myers, Citrus: a language and toolkit for simplifying the creation of structured editors for code and data, in: Proceedings of the 18th annual ACM symposium on User interface software and technology, ACM, Seattle WA USA, 2005, pp. 3–12. URL: https://dl.acm.org/doi/10.1145/1095034.1095037. doi:10.1145/1095034.1095037.

[11] B. Myers, R. McDaniel, R. Miller, A. Ferrency, A. Faulring, B. Kyle, A. Mickish, A. Klimovitski, P. Doane, The Amulet environment: new models for effective user interface software development, IEEE Transactions on Software Engineering 23 (1997) 347–365. URL: http://ieeexplore.ieee.org/document/601073/. doi:10.1109/32.601073.