Binding Orthogonal Views for User Interface Design

Olivier Beaudoux, Mickael Clavreul, Arnaud Blouin

Binding Orthogonal Views for User Interface Design

Olivier Beaudoux[1,2], Mickael Clavreul[1]
[1]ESEO
TRAME
Angers, France
{Olivier,Mickael}.{Beaudoux,Clavreul}@eseo.fr

Arnaud Blouin[2]
[2]INSA Rennes
TRISKELL (INRIA-IRISA)
Rennes, France
{blouin,beaudoux}@irisa.fr

ABSTRACT

The design of graphical user interfaces (GUI) is a complex activity that binds multiple concerns into meaningful interactive applications. While the definition of multiple orthogonal views greatly facilitates the design of such GUIs, existing GUI frameworks still lack a proper strategy for splitting and binding these views to build complete GUIs. This paper proposes a domain specific language (DSL) called Loa to unify both the definition of orthogonal views and the bindings between these views in the context of GUI development. We present the overall strategy for splitting and binding views based on the use of the Loa DSL and we illustrate the proposed approach on the design of a web-based application.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE), User interfaces*

Keywords

Domain Specific Language (DSL); Orthogonal Views; Reactive Programming; Active Operations; Graphical components; Graphical User Interface (GUI)

1. INTRODUCTION

The design of graphical user interfaces (GUI) is a complex activity that binds multiple concerns into meaningful interactive applications. The separation of the design of applications into multiple orthogonal concerns (*e.g.* domain objects, presentation, interaction, action) facilitates the design of GUIs. While existing modern GUI frameworks clearly separate the domain objects from their presentations, the definition of interactions and actions and the binding interaction-action is still sporadic. Existing GUI frameworks also require manipulating various languages to address each concern separately. Therefore, the proper decomposition of concerns is still incomplete while the corresponding binding activity is not easily tackled, thus hindering the initial goal of simplifying the implementation of GUIs.

Orthographic Software Modeling (OSM) targets the automatic generation of an application by decomposing a model into orthogonal views [1, 2]. Each view relates to a specific concern thus simplifying the construction of the complete model of a software. A common metaphor of OSM views is the decomposition of a 3D object along the three 2D plans $(0xy)$, $(0xz)$ and $(0yz)$. These projection views give a representation of the 3D object that eases its overall perception. Orthogonal projections do not necessarily mean that projections are completely independent. For instance, axis $(0x)$ represents the intersection of both the views $(0xy)$ and $(0xz)$ thus binding them against the coordinate x.

This paper proposes a conceptual model to unify both the decomposition of GUI applications into four orthogonal views and the bindings between these views. The conceptual model is formalized with a single Domain Specific Language (DSL) called Loa.

Section 2 discusses the work related to the definition of multiple views and their bindings in the context of GUIs. Section 3 presents the four views and six bindings that compose the Loa conceptual model. Section 4 details the main constructs of the Loa DSL that supports the conceptual model. Section 5 concludes this work and proposes perspectives.

2. RELATED WORK

Conceptual models that target the decomposition of an application into multiple orthogonal concerns are multiple. We focus on the most representative conceptual models regarding the current GUI frameworks available: MVC [3], MVC2 [4, 5] and the Interactor Model [6].

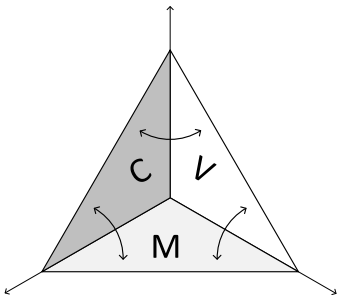

Figure 1: Orthogonal views within MVC

The Model-View-Controller (MVC) is a software architecture pattern that decomposes interactive software into three orthogonal views. Figure 1 illustrates the three orthogonal views represented as triangles, and the interaction links represented as arrows. Model *M* defines the *domain objects*, view *V* defines a *presentation* of the

domain objects, and controller C defines how the domain objects and the presentation change when the users interact with the system. The MVC interaction links follow a typical flow : (1) the views register as observers of the model and retrieve data to build a first presentation (link $V \rightarrow M$); (2) user interactions are captured by the controller that modifies the model (link $C \rightarrow M$) which in turn, notifies the views (link $M \rightarrow V$); (3) the views update depending on model changes (link $V \rightarrow M$); (4) Special user interactions modify a view without modifying the model (*e.g.* reporting an invalid input value): the controller modifies its associated views (link $C \rightarrow V$). Link $V \rightarrow C$ denotes that the view and the controller are paired, while link $M \rightarrow C$ is equivalent to $M \rightarrow V$ followed by $V \rightarrow C$; both $M \rightarrow C$ and $V \rightarrow C$ are useless regarding the flow.

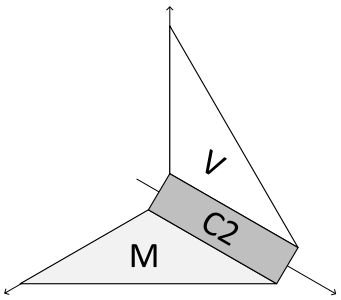

Figure 2: Orthogonal views within MVC2

While MVC isolates each part of the software, the definition of complex relationships between models and views is still complex using the links $V \rightarrow M$ and $M \rightarrow V$. MVC2 proposes an evolution of MVC to tackle this issue (see Figure 2). Controller *C2* represents the bidirectional link $M \leftrightarrow V$ and view *V* represents the paired controller and view of MVC. Rich Internet Application toolkits propose data binding mechanisms that greatly simplify the implementation of the *C*ontroller *C2* [7, 8]. The evolution of the controller from an orthogonal view to a view link and the use of data binding mechanisms highlight the importance of the orthogonal view links.

MVC and MVC2 mainly focus on separating the domain objects from their presentation but still lack in providing mechanisms for separating the interactions on presentation from actions on the domain objects.

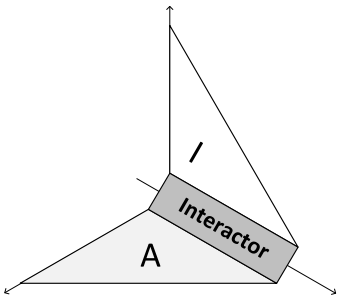

Figure 3: The Interactor Model

The Interactor Model explicitly separates the interaction objects and the action objects to propose two views bound by interactors (see Figure 3). An interactor is a specific object that transforms a user interaction into an action that has an effect on the presentation and/or on the domain objects.

While the selected strategies isolate software concerns using orthogonal views, the production of consistent and maintainable GUIs is still complex: (1) the mechanisms for binding the four orthogonal views (*i.e.* domain objects, presentation, interaction, and action) are not provided, thus requiring the knowledge of an expert; and (2) each orthogonal view uses its own formalism and its own linking definition, thus hindering an easy and smooth integration. This paper proposes a conceptual model to unify both the decomposition of GUI applications into four orthogonal views and bindings between these views using a homogeneous language (*i.e.* a DSL) based on usual OOP constructions.

3. THE LOA CONCEPTUAL MODEL

Figure 4 illustrates the "Graf" application dedicated to the editing of simple graphs using three tools: the *Node*, the *Link* and the *Rubber* tools. This application is the running example for presenting the Loa conceptual model detailed in this section and the Loa DSL presented in Section 4.

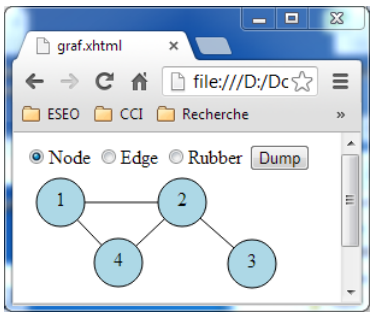

Figure 4: Screenshot of the "Graf" application

Figure 5 introduces the Loa conceptual model as a four-dimension model represented as an unfolded tetrahedron with four orthogonal views as triangles and six bindings as thin rectangles. The orthogonal views consist in the domain *objects*, the graphical *templates* (*i.e.* presentation), the *interactions*, and the *actions*. The bindings consist in the concepts of *mapping*, *interactor,* and *picking*.

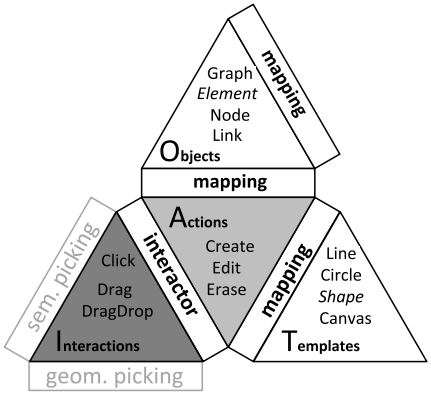

Figure 5: Loa conceptual model instantiated for the "Graf" application

The Graf application defines the *Graph*, *Node,* and *Link* objects of the domain, with respectively the *Canvas*, *Circle,* and *Line* graphical templates for displaying the domain objects to users.

While the domain objects and graphical templates are defined depending on the target application, actions and interactions are selected from a predefined set of interactions/actions provided by the Loa DSL: (1) the *Node* tool uses the *Click* interaction along with the *Create* action for creating new nodes; (2) the *Link* tool uses the *DragDrop* interaction along with the *Create* action to create links between two nodes, and (3) the *Rubber* tool uses the *Click* interaction along with the *Erase* action for erasing nodes or links.

We express relationships between actions, objects, and templates with mappings. The role of a mapping between objects of the domain and graphical templates is similar to the controller *C2* of MVC2: a *Graph* object is mapped to a *Canvas* template and a *Node* object is mapped to a *Circle* template to create the proper presentation of the domain objects. A mapping between an action and an object of the domain defines the effect of the action onto the object: a *Create* action mapped with the *Node* tool states that a *Node* instance is created and inserted into a *Graph* instance. Special mappings between an action and a template allow the application to give feedback to users with no changeon the domain objects.

An interactor transforms an interaction in an action: transforming the *Click* interaction in the *Create* action defines the behavior of the *Node* tool. The interactor thus becomes a manipulable object that specifies how to mix the interaction with the action effect.

Pickings represent the ability to select an object at a given coordinate. *Geometric picking* (respectively *semantic picking*) represents the link between the interaction and the graphical template (respectively the domain object). The *Click* interaction of the *Node* tool specifies that (double) clicking on a *Graph* instance triggers the creation of a new node.

In Section 4, we detail the Loa DSL on the Graf application example. We demonstrate how the Loa DSL supports the definition of the four orthogonal views and how it helps to bind those views with one another, following the conceptual model presented in this section.

4. THE LOA DSL

The Graf application (see Figure 4) runs on a HTML5 / JavaScript platform. The entry point of the application is defined as an HTML document as follows:

```
1  <html>
2    <script src='loa.js '/>
3    <script src='graf.js '/>
4    <body onload='graf.main();'>
5      <input type='radio' id='nodeBtn' name='tool'
6        checked='true'/>
7      Node
8      <input type='radio' id='edgeBtn' name='tool'/>
9      Edge
10     <input type='radio' id='rubberBtn' name='tool' />
11     Eraser
12     <div id='scene'/>
13   </body>
14 </html>
```

This snippet contains the static and initial contents of the Graf application. It includes the Loa API "loa.js" (line 2) and the JavaScript code "graf.js" (line 3) generated from the Loa model designed for the Graf application. Dynamics of the application, which populate the contents of the * * element (line 12), are provided by the Loa DSL in the "graf.loa" model. We detail the production of the model in the Section 4.

4.1 Definition of views with Loa

We define the four orthogonal views of the Graf application using the constructs of the Loa DSL [9].

View 1: Domain Objects

The Loa DSL proposes four primitive types *int*, *float*, *boolean,* and *string* to characterize literal values. Six types of observable container (*i.e. one*, *opt*, *set*, *oset*, *seq,* and *bag*) are available to define the properties of each domain object. Observable container types support the definition of bindings between the orthogonal views [10, 11], providing a well-defined set of operations.

Observable containers qualify both the minimal cardinality (0 for all containers but *one*), the maximal cardinality (1 for *opt* and *one*, unbounded otherwise), and additional *unique* and *ordered* constraints (similar to OCL collections).

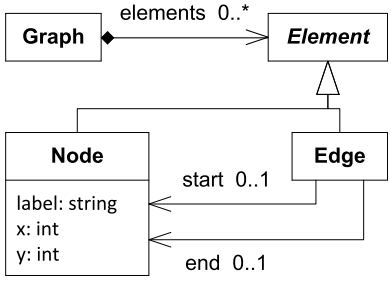

Figure 6: The domain objects as a UML class diagram

Figure 6 illustrates the representation of the domain objects of the Graf application using a UML class diagram. The following listing illustrates how we model the domain objects with the Loa DSL:

```
class Graph {
  set<Element> elements;
}

abstract class Element { }

class Node extends Element {
  one<string> label;
  opt<int> x;
  opt<int> y;
}

class Edge extends Element {
  opt<Node> start;
  opt<Node> end;
}
```

A *Graph* defines its *elements* as a **set** containing *Element*s. A *Node* is an *Element* that defines **one** mandatory textual *label*, and two **opt**ional coordinates *x* and *y* that contain the node location once the graph has been laid out. An *Edge* is an *Element* that defines two **opt**ional edges *start* and *end* denoting the current connections to nodes. As one may note, the specification of domain objects is platform-independent.

View 2: Graphical Templates

The definition of the presentation view is decomposed into templates. Templates define the graphical components that compose the view. The graphical templates are instantiated at runtime to populate the graphical representation of the application [12].

Figure 7 illustrates the definition of the presentation view for the Graf application using a UML class diagram. The template class *Canvas* represents the root of the graph displayed as a SVG image. *Canvas* defines two layers *topLayer* and *bottomLayer* that can contain *Shape*s. A *Shape* is an abstract class that defines a position through coordinates *x0* and *y0*, and a *stroke* color. The

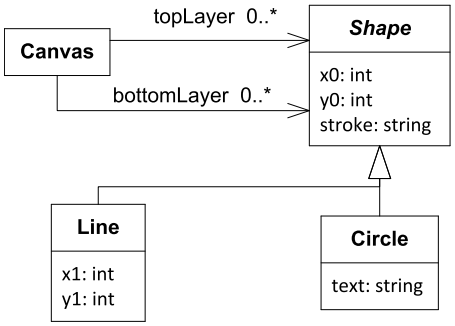

Figure 7: The graphical templates as a UML class diagram

template class *Line* adds a second point with coordinates *x1* and *y1* to draw a line from coordinate (*x0,y0*) to coordinate (*x1,y1*). The template class *Circle* adds an optional short *text* to be displayed if necessary. The following listing illustrates how we model the template classes with the Loa DSL:

```
template class Circle extends Shape {

    opt<string> text;

    <g transform="translate({x0},{y0})">
        <circle  r="20" stroke="{stroke}"
             fill ="lightblue"/>
        <text  y="3"  text-anchor="middle">
            {text}
        </text>
    </g>

}
```

Graphical templates as Loa classes augmented with the concrete production of the graphical components. For example, the *Circle* template class defines four properties *x0*, *y0*, *stroke,* and *text* associated with a SVG group <*g*> that contains two elements <*circle*> and <*text*>. Expressions in brackets define the variation points within the template that are directly bound to the associated properties. In the *Circle* template class example, the location of the SVG group depends on the two properties *x0* and *y0* from a given *Shape* as well as the *stro*ke color of the *text* contents. Details about the instantiation of the graphical templates is out of the scope of this article. Readers can refer to [12] for more details.

In this section, we illustrate the use of the graphical templates with SVG components to target Web platforms and especially HTML5 presentations. Other implementations target Java platforms [9]. The Loa DSL only imposes that the templates are written in XML.

View 3 & 4: Interactions and Actions

Actions and interactions are Loa classes that represent the available interactions and actions supported by the target platform. The list of actions and interactions is thus *predefined* within the Loa DSL.

The following listing illustrates how we model the *Click* interaction with the Loa DSL:

```
class Click {
    one<int> button = 1;
    one<int> clickCount = 1;
    opt<int> x;
    opt<int> y;
    one<Class> clickableClass;
    opt<Object> clickedObject;
    opt<Object> clickedTemplate;
}
```

Properties *button* and *clickCount* customize the *click* interaction. Properties *x* and *y* capture the location of the mouse cursor where events occur. The three last properties relate to semantic and geometric pickings (see Section 3). Property *clickableClass* indicates which domain objects are clickable. Property *clickedTemplate* represents both the clicked object instance of this clickable class (semantic picking) and the clicked template that displays the clicked object (geometric picking).

The following listing illustrates how we model the *Create* action with the Loa DSL:

```
class Create {
    one<Property> creationRelation;
    one<Class> creationClass;
    opt<Object> creationParent;
    opt<Object> createdObject;
}
```

When mixed with class *Click*, the class *Create* allows the definition of an interactor that creates new nodes when the user double-clicks on the graph. Details are presented in Section 4.2.

4.2 Binding views with Loa

The following sections detail mapping and interactor bindings discussed in Section 3.

Binding 1: Mappings

Figure 8 illustrates the mappings between the domain objects and the graphical templates with a UML representation augmented with mappings as named circles. Three mappings *G2C*, *L2L,* and *N2C* map the domain objects to the corresponding graphical templates.

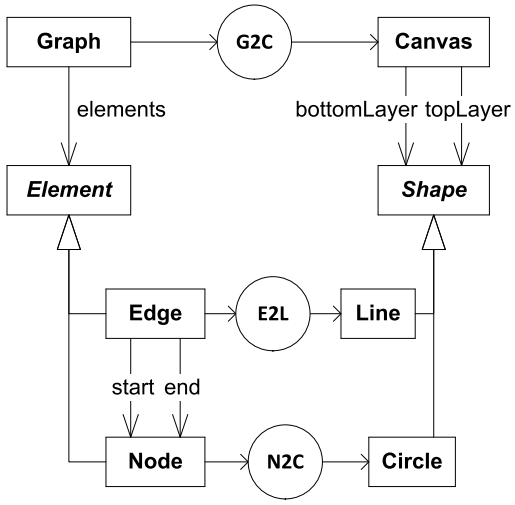

Figure 8: Mappings between the domain objects and the graphical templates

With Loa, we implement these mappings as functions that use active operations as follows:

```
1  Canvas G2C(Graph g) {
2      Canvas c = new Canvas();
3      c.topLayer := g.elements.as(Node).map(N2C);
4      c.bottomLayer := g.elements.as(Edge).map(E2L);
5      return c;
6  }
7
```

```
8  Line E2L(Edge e) {
9      Line ln = new Line();
10     ln.x0 := e.start.x;
11     ln.y0 := e.start.y;
12     ln.x1 := e.end.x;
13     ln.y1 := e.end.y;
14     return ln;
15 }
```

Operation *as* in mapping *G2C* selects all instances of class *Node* from a graph *g*. Operation *map* applies the mapping function *N2C* to each selected nodes (line 3). Operator *:=* denotes an "active" assignment that synchronizes the contents of property *c.topLayer* with the contents of property *g.elements* at runtime (line 3). Line 4 follows the same construction for populating the bottom layer with the edges created by the mapping function *E2L*. Each edge is represented as a *Line* with coordinates given by the coordinates of the *start* and *end* nodes of the edge (lines 9 to 13).

Binding 2: Interactors

Binding interactions with actions is implemented as class mixins represented with the *&* operator in the Loa DSL. For example, the mixing of a *Click* interaction with a *Create* action defines the "node" interactor that allows the construction of a new *Node* as the result of a *Click* event. The following listing illustrates how we model such an iteractor with the Loa DSL:

```
1  Click&Create NC(RadioButton nodeBtn) {
2      Click&Create i = new Click&Create();
3      i.enabled := nodeBtn.selected;
4      i.clickCount = 2;
5      i.clickableClass = Graph;
6      i.creationClass = Node;
7      i.creationRelation = Graph.elements;
8      opt<Node> createdNode = i.createdObject.as(Node);
9      createdNode.x := i.x;
10     createdNode.y := i.y;
11     return i;
12 }
```

The resulting interactor *i* is created on line 2. This interactor is enabled only if the associated radio button *nodeBtn* is selected (line 3). We filter *Click* interactions on the source on the event to prevent creating dangling objects in the model of the domain (line 5). We specify which property is going to be populated with the new instance (line 7), build an instance as a node (line 8), and provide its position (lines 9 and 10).

4.3 Running the example

In order to bind the four orthogonal views together, we define a Loa application as a set of mappings and interactors. The next listing illustrates how the Loa DSL bridges the gap between the bound orthogonal views and the execution platform to produce the Graf application:

```
1  void main() {
2      opt<Graph> graph;
3      page.scene := graph.map(G2C);
4      page.interactors := page.nodeBtn.map(NC) ++
5                          page.nodeBtn.map(NM) ++ ...;
6      graph = sampleGraph();
7  }
```

Variable *graph* represents the graph model, initially empty (line 2), of the Graf application. Variable *page* is a Loa keyword for accessing the current edition page (*i.e.* the HTML document presented in Section 4). Property *scene* (line 3) represents the <*div*> tag and property *nodeBtn* (line 4) represents the first <*input*> tag of the current edition page. We associate the <*div*> tag of the editor with the canvas created by mapping G2C (line 3) and we register the available interactors (line 4 and 5). Line 6 populates the initial graph with the Graf application example.

5. CONCLUSION AND PERSPECTIVES

This paper presents a conceptual model to unify both the decomposition of GUI applications into four orthogonal views and the bindings between these views. The conceptual model is supported by the Loa DSL, an homogeneous language that eases the definition of views and their bindings. A reasonably simple example illustrates how the Loa DSL clarifies the development of GUIs. Available implementations of the Loa DSL targets both Java and HTML5 platforms and will be available as open source projects shortly.

We expect the Loa DSL to be proficient in targeting groupware and constraint management concerns in future work. Supporting additional concerns may lead to improving the Loa DSL for dealing with the definition of new views and their bindings.

6. REFERENCES

[1] Atkinson, C., Stoll, D., Bostan, P.: Supporting view-based development through orthographic software modeling. In: Proc. of ICENAS'09, Springer (2009) 71–86

[2] Atkinson, C., Bostan, P., Brenner, D., Falcone, G., Gutheil, M., Hummel, O., Juhasz, M., Stoll, D.: Modeling components and component-based systems in kobra. LNCS 5153 (2008) 54–84

[3] Krasner, G.E., Pope, S.T.: A description of the model-view-controller user interface paradigm in smalltalk80 system. Journal of Object Oriented Programming **1** (1988) 26–49

[4] Seshadri, G.: Understanding javaserver pages model 2 architecture. JavaWorld (99)

[5] Foundation, A.: Struts. Online (04 2013)

[6] Myers, B.A.: A new model for handling input. ACM TIS'99 **8**(3) (1990) 289–320

[7] Dewsbury, R.: Google Web Toolkit Applications. Addison-Wesley Professional (2008)

[8] Sells, C., Griffiths, I.: Programming Windows Presentation Foundation. O'Reilly (2005)

[9] Beaudoux, O., Clavreul, M., Blouin, A., Yang, M., Barais, O., Jezequel, J.M.: Specifying and running rich graphical components with loa. In: Proc. of EICS'12. (2012) 169–178

[10] Beaudoux, O., Blouin, A., Barais, O., Jezequel, J.M.: Active operations on collections. In: Proc. of MoDELS '10, Springer (2010) 91–105

[11] Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.M.: Specifying and implementing ui data bindings with active operations. In: Proc. of EICS'11, ACM (2011) 127–136

[12] Beaudoux, O.: XML active transformation (eXAcT): Transforming documents within interactive systems. In: Proc. of DocEng'05, ACM (2005) 146–148