Specifying and Running Rich Graphical Components with Loa
Olivier Beaudoux, Mickaël Clavreul, Arnaud Blouin, Mengqiang Yang, Olivier Barais, Jean-Marc Jézéquel

To cite this version:

HAL Id: hal-00684881
https://inria.hal.science/hal-00684881
Submitted on 18 Jul 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Specifying and Running Rich Graphical Components with Loa

Olivier Beaudoux\textsuperscript{1}, Mickael Clavreul\textsuperscript{1}, Arnaud Blouin\textsuperscript{2}, Mengqiang Yang\textsuperscript{1}, Olivier Barais\textsuperscript{2}, Jean-Marc Jezequel \textsuperscript{2}

\textsuperscript{1}ESEO, TRAME Team
LUNAM University
Angers, France
\textsuperscript{2}University of Rennes 1
IRISA, Triskell Team
Rennes, France
[first-name].[last-name]@eseo.fr [last-name]@irisa.fr

**ABSTRACT**
Interactive system designs often require the use of rich graphical components whose capabilities go beyond the set of widgets provided by GUI toolkits. The implementation of such rich graphical components require a high programming effort that GUI toolkits do not alleviate. In this paper, we propose the Loa framework that allows both the specification of rich graphical components and their integration within running interactive applications. We illustrate the specification and integration with the Loa framework as part of a global process for the design of interactive systems.

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

**General Terms**
Algorithms, Design, Languages

**Author Keywords**
Graphical components; Graphical User Interface (GUI); Domain Specific Language (DSL); Active Operations

**INTRODUCTION**
Building Rich Interactive Applications (RIA) often leads to the design of new graphical components. Recent interactive systems such as smart-phones and tablets well illustrate such a purpose. The design of complex graphical components that goes beyond the composition of existing widgets requires implementation effort and prevents reuse or capitalization. While recent GUI toolkits, such as GWT\textsuperscript{26}, WPF\textsuperscript{22}, and Flex\textsuperscript{14}, target the implementation of RIAs, the design of rich graphical components is still a time-consuming activity.

Providing the right methodologies, models and tools that facilitate the design and the implementation of such rich graphical components is thus an important challenge. Standard GUI toolkits are centered on a low level implementation of graphical components and are based on primitive drawing functions. This statement applies to proven toolkits such as Swing\textsuperscript{9} and remains for recent standards such as HTML 5\textsuperscript{10}. This situation leads to interoperability and synchronization issues between the graphical design made by the designers and its implementation written by the programmers. Current RIA environments solve this issue by proposing tools that integrate (or link) both the design and the programming environments. For instance, FlexBuilder is both a designing and programming environment dedicated to Flex applications; Expression Blend is a designer environment that produces XAML documents that can be directly used within VisualStudio, a programming environment.

Integrating design and programming environments provides interoperability and synchronization to some extent. But in this case the specification of new graphical components requires a significant implementation effort. Effort includes the definition of components, the implementation of the associated interactions and the related actions performed on the domain data, and their integration within the final application.

In this paper, we propose the Loa framework that allows both the specification of rich graphical components and their integration within running interactive applications. This framework homogenizes the process of extending and integrating previous works on data binding\textsuperscript{3, 4}, graphical templates\textsuperscript{13, 27, 2}, and interactors\textsuperscript{19, 1, 6}. The paper focuses on the specification and integration with the Loa framework as part of a global process for the design of interactive systems.

The remainder of the paper is structured as follows. Next section presents a global overview of the Loa framework that includes a DSL (Domain Specific Language), a development process, and a supporting tool implementation. Sections “Step 1” to “Step 6” detail each step of the process through the concrete example of building a planner application. We discuss this work and evaluate Loa with regard to existing GUI toolkits in Sections “Rela-
A Scala-based DSL

The Loa framework is based on the Scala language [20]. This choice is motivated by the following facts: 1) Scala is an OO language fully compatible with Java, thus allowing the reuse of the numerous Java API and tools, especially GUI ones; 2) Scala embeds the imperative the functional and the object-oriented programming paradigms in a way that greatly facilitates the implementation of the framework; 3) Scala is a self-extensible language that enables the construction of new DSLs while reusing the infrastructure and tools (i.e., IDEs and compilers) of the Scala language for these DSLs.

The Loa DSL is thus built as an extension of the Scala language that defines the very language of the framework.

With this short introduction in mind, the Loa DSL allows designers to capture the problem domain of GUI engineering within a precise, concise and dedicated language. This paper focuses on the usage of the Loa framework supported by the Loa DSL in the process of building new graphical components for interactive systems. Details on the Loa DSL are thus out of the scope of the paper; the DSL is rather explained through a concrete example.

The development process

The Loa framework splits the development process of interactive systems into six steps as illustrated in Figure 1. Using the usual concept of classes, application designers specify the domain data (step ①). Graphical designers sketch new graphical components using a graphical design tool such as Illustrator or Inkscape (step ②) that produces an XML document. Sketches are then formalized into graphical templates (step ③) that both define the graphical parts of the components in XML and their parameters. While steps ①, ②, and ③ require interaction between application and graphical designers, they can be executed in parallel. Application designers specify the data binding (step ④) using the data binding capabilities of active operations ③④. Data binding consists in linking the domain data to the graphical templates. Application designers specify the interactors (step ⑤) using the data binding capabilities provided by the framework and an interaction model inspired from Malai [6]. The specifications that result from steps ①, ③, ④ and ⑤ are finally integrated together in a sole executable application. While the execution of the resulting application requires the whole set of specifications, each specification can be tested independently. For instance, a given graphical template can be tested while the specification of domain data or interactors are not yet provided.

While the process is proposed for building new graphical components, the use of existing widgets provided by GUI toolkits fits in. In such a case, step ② relies on GUI design tools, such as Adobe FlexBuilder, Microsoft Expression Blend or Google WindowBuilder. Step ③ and ④ thus consists in reusing the Loa model of these widgets. Steps ①, ④ and ⑥ remain unchanged.

As one may note, the process focuses on the idea of specification. The ability to run a specification without requiring a hand-written implementation of the specification is an essential contribution that we summarize as:

“Specifying GUIs rather than implementing them”

However, the process is not a methodology for the development of interactive systems. It is rather a process that formalizes the use of the Loa framework; it can be used jointly to well-known methodologies such as user’s task analysis [21].

The implementation

The implementation of the Loa framework consists in an API that bridges the three main concepts of data bindings, of graphical templates, and of interactors with existing GUI toolkits.

The current implementation provides a bridge to Swing widgets as well as Batik for the definition of graphical templates based on the SVG standard. The case study that we present in the paper use the latter. The case study has been built using Inkscape for sketching the graphical templates, using Google WindowBuilder for designing the Swing UI in a WYSIWYG manner, and using the Eclipse workbench with Scala support for the edition and the compilation of the code written in Loa.
STEP 1: SPECIFY THE DOMAIN DATA
Figure 2 gives the class diagram that represents the domain data of the academic planning: Planning of a given year is composed of 0 to 52 weeks; a teaching Week is composed of 0 to 5 days; and each Day defines its teachings. A Teaching can span among multiple TimeSlots: the first element of relation timeSlots defines the starting time-slot, while the second element gives the ending time-slot. A Teaching also references a topic, a teacher, and a room.

![Class diagram of an academic planning](image)

Figure 2. Class diagram of an academic planning

The Loa data model provides a textual representation of classes based on the use of six kinds of Loa container. This data model is used for specifying the domain data, the graphical templates, as well as the interactors. The following listing illustrates the specification of the class Planning from the domain data using the three Loa containers One, OSet and Set:

```scala
class Planning {
  val year = One(2012)
  val weeks = OSet[Week]()
  val resources = Set[Resource]()
}
```

Attribute year is represented as an integer boxed into a container with initial value 2012. Relation weeks is represented as an initially empty ordered set of Week instances, and relation resources as a set of Resource instances.

<table>
<thead>
<tr>
<th>Container</th>
<th>Cardinality</th>
<th>Uniqueness</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt</td>
<td>0..1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One</td>
<td>1..1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bag</td>
<td>0..*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq</td>
<td>0..*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>0..*</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>OSet</td>
<td>0..*</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 1. Loa container kinds

Container contents can be modified using assignment operators. Operator := is used for Opt and One containers, and operators += and -= are used for the four collection containers, as follows:

```scala
val p = Planning()
p.year := 2012
p.weeks += Week()
```

All the six kinds of container implement an observability mechanism that allows each change (addition, removal or update) to be captured. Observability is an important feature for graphical components, data bindings and interactors.

Finally, methods written in Scala define the application logic that is integrated in the classes of the system.

STEP 2: SKETCH THE GRAPHICAL COMPONENTS
Figure 3 gives a screen-shot of the Academic Planning Application that displays a weekly view of the academic planning, as introduced in the previous section; each teaching is presented to users through a stamp graphical component. Menu File allows loading and saving XML documents that contain planning data related to a specific academic year. Menu Edit allows editing planning resources (e.g., rooms, topics), and allows duplicating or clearing the week contents. The combo-box at the top allows selecting a specific week within a year (from 1 to 52). The canvas allows direct manipulation of the selected week. Undo and redo buttons allow undoing and redoing past actions. Finally, the tab Tree displays the tree of the graphical scene for debugging purpose.

![Academic Planning Application](image)

Figure 3. The Academic Planning Application

Based on the application requirements, the graphical designer sketches the necessary graphical components exported by the tool as XML representations. The following code illustrates an excerpt of a stamp component
that uses the SVG format for its XML representation:

```xml
<g transform="translate(10,10)">
  <rect width="170" height="70" fill="white"/>
  <text x="-5" y="11" transform="rotate(-90)" ...>
    Course
  </text>
</g>

The *stamp* is an SVG group composed of: a white background (line 3); a type bar representing the *stamp* type with a text (lines 6 to 8) and a yellow background (line 5); a group of labels displaying the event description (lines 11 to 13); and a black stroke defining the stamp border (line 16 and 17).

Since SVG does not support the parameterization of symbols, the code of the *stamp* graphical component includes hard-coded value. Step ➂ addresses such parametrization issues.

**STEP 3 : SPECIFY THE GRAPHICAL TEMPLATES**

Loa formalizes sketches of graphical components (step ➂) as the combination of a Loa data model and templates. In the following subsections, we present the specification of such templates with Loa and we explain how templates are transformed into an executable form.

**Specification of templates**

Figure 4 shows the class diagram which contains the *template classes* that formalize the sketched graphical components used by the planning application. The root template class *Calendar* represents the weekly presentation of a planning.

The presentation of a calendar is a grid that is composed of *dayLabels* and *timeAreas*. Its contents is displayed by *stamps*, each *Stamp* including a *typeBar* (*i.e.*, the type of the displayed event) and *labels*.

A template class is specified using both the Loa data model and the specification of its *template*. Template class *Stamp* specifies attributes *x*, *y*, *width* and *height* and relations *typeBar* and *dayLabels* as follows:

```scala
class Stamp extends Fragment {
  val x = One(0)  val width = One(300)
  val y = One(0)  val height = One(100)
  val typeBar = One(TypeBar())
  val dayLabels = OSet[Label]()
}
```

Loa introduces the keyword *template* defined by class *Fragment* (term fragment is explained in the next subsection). Parametrization of a template is carried out by the concept of *anchor*: an anchor is bound to a container that defines the anchor contents and the anchor location within the template through symbol §. In the *Stamp* example, anchors §x, §y, §width and §height (lines 8 and 9) receive the contents of their associated containers *x*, *y*, *width* and *height* (lines 2 and 3); anchor §typeBar (line 11) receives the associated *TypeBar* instance contained in *typeBar* (line 4); anchor §dayLabels (line 13) will receive multiple *DayLabel* instances subsequently added into the ordered set *dayLabels* (line 15).

**Instantiation of templates**

Instantiation of a Loa template into a fragment consists of creating a DOM fragment with an initial content defined by the template itself, and binding the anchors of the templates within the DOM fragment. The interlacing between fragments and anchors allows the incremental construction of the final graphical components. The interlacing approach has been borrowed from eXAcT [2]. Since the mechanism is quite complex, we provide an overview of the core principles. Figure 5 illustrates the
instantiation of the template $Stamp$ into a fragment. For the sake of clarity, Figure [5] only contains the $width$, $height$ and $dayLabels$ anchors.

Figure 5. A subset of an instance of fragment $Stamp$

Instantiation of the template $Stamp$ consists of creating a DOM fragment containing a root $<g>$ element and all its nested nodes until an anchor specification is reached. For instance, when attribute $width$ of element $<rect>$ is reached, an anchor is created. This anchor observes the value of its associated container $width$ defined in the enclosing fragment class. Similarly, when child $<g>$ element is reached, an anchor is created. The content of this anchor is delimited by two processing instruction $<?begin ?>$ and $<?end ?>$ that reflects the container $dayLabels$ : adding a new instance of class $DayLabel$ to the relation $dayLabels$ triggers the corresponding template (containing element $<text>$). The corresponding template is thus instantiated and the resulting child fragment is inserted in between the delimiters. Separation of subsequent $DayLabel$ fragments is represented by the processing instruction $<?next ?>$.

STEP 4 : SPECIFY THE DATA BINDING

Besides the fact that Loa containers are observable, the Loa data model allows binding two containers so that their contents remains the same. Loa introduces the assignment operator $: :=$ so that expression $a : := b$ means that container $a$ receives the same contents as container $b$. When used concurrently with operations available in the Loa DSL (e.g. $apply$), the assignment operator makes possible to bind multiple containers, as illustrated by the following example :

1 val a = Seq[Int]()
2 val b := a.apply(e => e.toString())
3 a += 123

This example binds the sequence of integers $a$ to the sequence of strings $b$. Line 2 means that $b$ contains the string representation of integers contained in $a$. Since $a$ is initially empty, $b$ is also initially empty. When the sequence $a$ is modified on line 3, the sequence $b$ is updated by the execution of operation $apply$ accordingly, such that $b$ finally contains “123”. The implementation of the function $apply$ is based on the observation of sequence $a$ and on the execution of the anonymous function $e => e.toString()$ in an appropriate manner. More details on the implementation of complex functions such as $select$ and $sort$, is detailed in [3].

Binding containers allows the specification and execution of complex data bindings that bind the domain data to graphical components [4]. The instantiation of templates is thus driven by the changes performed on the domain data. As an illustration, the following function $P2C$ is a Loa specification that represents the data binding between a planning $p$ loaded from menu $File$, a combo-box $ws$ for selecting the current week, and a calendar $c$ that gives a weekly presentation of $p$ to users (see Figure [3]) :

```scala
def P2C(p: Planning, ws: ComboBox, c: Calendar) = {
  c.dayLabels ::= Seq(0 to 4).apply(DN2DL)
  c.timeAreas ::= p.timeSlots.map(TS2TA)
  ws.items ::= p.weeks.map(W2I)
  c.stamps ::= ws.selectedItem.rmap(W2I)
  .days.teachings.map(T2S)
}
```

From a static sequence of day numbers (0 to 4), the binding function $DN2DL$ creates the fragments of anchor $dayLabels$ (line 2). Line 3 defines the contents of anchor $timeAreas$ using the binding function $TS2TA$ on $timeSlots$. Line 4 populates the $ws$ combo-box with the planning weeks by calling the mapping function $W2I$ : the selected item of $ws$ drives the creation of $Stamp$ fragments within anchor $stamps$ retrieves the selected week (function $rmap$, line 5) and applies the mapping function $T2S$ to all teachings of this week.

The contents of mapping functions $DN2DL$, $TS2TA$, $W2I$ and $T2S$ follows the same constructs as $P2C$. This example shows that we use equivalent mapping constructs for both existing widgets (e.g., Swing combo-box), or specific graphical components (e.g., an SVG graphics).

STEP 5 : SPECIFY THE INTERACTORS

The Loa interactors are based on Malai [6]. An interactor transforms an interaction into an action on the target object. Within Loa, the target object can be either a domain data or a fragment.

Figure 6. Action life-cycle
Figure 6 illustrates the life-cycle of the action of any Loa interactor as a generic state-machine. For a mouse-based interactor, each transition corresponds to a mouse event as follows: begin = mouse button pressed, do = mouse dragged, abort = escape key pressed, end = mouse button released. If the interactor defines undo/redo data, these data are saved into the undo-redo stack. An interactor, such as a key or button, can request an undo or a redo, as illustrated at the end of this section. Finally, when the undo-redo stack reaches its maximal capacity, the undo/redo data are deleted.

Figure 7 shows a subset of the Loa interactor classes. Class Interactor defines the attribute canDo that indicates if the action associated with this interaction can be started or not. Class OneShotInteractor represents interactors with a state began that is immediately followed by a state end (e.g., Button and MenuItem). Conversely, class PointerInteractor represents interactors with a complex action life-cycle (e.g., Move and Click). Class Move defines properties (x0,y0) and (x,y) that represent respectively the starting mouse location and the current mouse location.

Class Interactor defines a Scala when block that allows designers to specify actions for each possible transition of the interactor, as follows:

```scala
when {
  case Begin =>
  case End =>
  case Abort =>
  case Undo =>
  case Redo =>
}
```

Each case corresponds to a transition and triggering a transition invokes the corresponding case block. The specification of actions for states depends on the kind of interactor. OneShotInteractor interactors only rely on the definition of either case block Begin or End. PointerInteractor interactors rely on case blocks Begin and End, and an optional case block Abort if the action needs to be aborted. Case blocks Undo and Redo provide undo/redo capabilities for OneShotInteractor and PointerInteractor interactors.

The following Hand interactor specifies how the user can move a stamp to another location:

```scala
object Hand extends Drag {
  when {
    case Begin(s: Stamp) =>
      cursor := Cursor.Move
      calendar.stamps += s // put s on the top
    val t = s.rmap(T2S)
    t.day ::= y.apply(y2Day)
    t.start ::= x.apply(x2TimeSlot)
    t.end ::= (s.width + s.x - x).apply(x2TimeSlot)
    undoData += (t, t.day, t.start, t.end)
  case End(s: Stamp) =>
    cursor := None
    val t = s.rmap(T2S)
    redoData += (t, t.day, t.start, t.end)
  case Undo(t: Teaching, d: Day, s: TimeSlot, e: TimeSlot) =>
    t.day ::= d
    t.start ::= s
    t.end ::= e
  }
}
```

Stamp s is given as a parameter in case blocks Begin and End, and corresponds to the stamp picked at location (x,y). Line 6 retrieves the teaching t that has been previously mapped to stamp s with the mapping function T2S, thanks to the reverse mapping operation rmap. Lines 7 to 9 define the new location of the picked teaching t by using layout functions that convert location (x,y) to Day and TimeSlot instances. The three properties day, start and end are bounded to the interactor location using operator :=, thus meaning that these three locations will be updated while the drag interaction continues. Case block Begin ends with saving undo data. Case block End terminates the action by resetting the cursor to its default value (line 12), then by saving the undo/redo data into the Loa undo stack (line 13 and 14). Based on the undo stack, case block Undo restores the saved state of teaching t when necessary (line 17 to 19).

We voluntarily omit case block Redo, since the undo/redo actions are based on the same data. Similarly, case block Abort(s : Stamp) has been omitted for simplification purpose.

Button undoButton (see figure 3) specifies its action in a simpler manner, as follows:

```scala
undoButton.when { case End => UndoStack.undo() }
```

---

3. Button redoButton is defined similarly.
Global object UndoStack represents the undo/redo stack used by interactors to store undo/redo data. Button undoButton binds its canDo attribute as follows:

undoButton.canDo ::= UndoStack.canUndo

The UndoStack object defines attribute canUndo that specifies whether the undo/redo stack contains undo data or not. This attribute is bound to the canDo attribute of the undo button, thus resulting in disabling or enabling the button depending on the stack state.

### STEP 6 : RUNNING THEM ALL TOGETHER

The following Scala code defines the main entry point of the application:

```java
object Application extends Frame {
  def main(args: Array[String]) {
    val calendar = Calendar()
    val svg = SVGScene(calendar, 1400, 600)
    val canvas = SVGCanvas(svg)
    val planning = Planning()
    planning.load("Planning 2011−12.xml")
    P2C(planning, weekSelector, calendar)
    canvas.interactors += Hand
    canvasScrollPane.setViewportView(canvas)
    treeScrollPane.setViewportView(DOMTreeView(svg))
    setVisible (true)
  }
}
```

The main window of the application is represented by a Java class Frame created with Google WindowBuilder. Since the integration of Scala code with Java is smooth, we may define a singleton class Application that extends Frame. Fragment calendar (line 3) is the root fragment of scene svg (line 4), which is rendered by the object canvas (lines 5 and 10) and displayed as a DOM tree view (line 11). Instance planning is loaded from an XML file (lines 6 and 7), and then bound to instance calendar with the application of function P2C (line 8). Combo-box weekSelector allows a user to select a week to edit (see Figure 11), and then bound to instance calendar with the display of the canvas and its components.

### RELATED WORKS AND DISCUSSION

Table 2 gives a synthetic comparison of related works for each of the four domains related to steps 1 to 5: the data model, the data binding, the graphical templates and the interaction model. Each domain is then evaluated against two or three criteria indicating whether: i) the data model allows the definition of observable properties and/or observable collections; ii) the data bindings can be defined for properties, collections using a dedicated language or not; iii) the graphical templates allow the definition of simple components (e.g., a TimeArea, a TypeBar - see Figure 4) and/or nesting components (e.g., a Calendar, a Stamp - see Figure 4); iv) the interaction model provides an event-based interaction model, an action model, and/or an interactor-based model. Values for criteria are: high (H), medium (M), low (L), and none (empty cell). Concepts of the related works that we reuse within Loa are in bold.

<table>
<thead>
<tr>
<th>Data model</th>
<th>Data binding</th>
<th>Gr. templates</th>
<th>Interaction model</th>
</tr>
</thead>
<tbody>
<tr>
<td>obs. prop.</td>
<td>prop. coll.</td>
<td>simple</td>
<td>events</td>
</tr>
<tr>
<td>obs. coll.</td>
<td>lang.</td>
<td>nesting</td>
<td>action</td>
</tr>
<tr>
<td>Swing</td>
<td>H</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>JFace</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>ObjectEditor</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Garnet</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>Amulet</td>
<td>H</td>
<td>H</td>
<td>M</td>
</tr>
<tr>
<td>Malai</td>
<td>H</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>JavaFX 2.0</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>Flex</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>JavaFX 1.3</td>
<td>H</td>
<td>H</td>
<td>M</td>
</tr>
<tr>
<td>WPF</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>Ext GWT</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Hayaku</td>
<td>M</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>inxXSLT</td>
<td>M</td>
<td>M</td>
<td>H</td>
</tr>
<tr>
<td>sXBL</td>
<td>M</td>
<td>M</td>
<td>H</td>
</tr>
<tr>
<td>eXAcT</td>
<td>M</td>
<td>M</td>
<td>H</td>
</tr>
<tr>
<td>Definitive pr.</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Active op.</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Loa</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>

Table 2. Synthetic View of Related Works
of each category.

**Java-based toolkits**

Swing [9] is based on the MVC pattern rather than on the data binding concept. Consequently, the definition of bindings requires a significant programming effort for implementing interfaces. Creating a new component is a time-consuming task that requires a full implementation from scratch. As for all GUI toolkits, the interaction model listens to predefined events. The concept of action is minimalist and does not integrate undo/redo features. JFace [12] explicitly uses data binding between observable properties, and allows binding observable collections with some predefined widgets. ObjectEditor [8] allows the definition of data bindings on properties and on collections, and enhances the action model with undo/redo capabilities. Since ObjectEditor is based on extending the JavaBean syntax, it might be considered as a language. We observe that the category “Java-based toolkits” both lacks mechanisms for the definition of graphical templates and of interaction model, and globally lacks dedicated languages for data binding. These issues are answered by the dedicated toolkits presented in the next sections.

**Interactor-based toolkits**

Garnet [17] and Amulet [18] explicitly introduce the concept of interactor to facilitate the use of predefined interactions. Amulet provides a high level action model with undo/redo/repeat features, as well as the ability to define simple graphical components easily. Garnet and Amulet allow the definition of constraints for binding properties with no support for collections. Malai [6] formalizes the instrumental interaction [1] into a well-polished conceptual framework. Malai allows binding collections but does not support complex computation on collections. Malai interaction model has been reused and adapted to fit in the Loa data model. We observe that while “Interactor-based toolkits” support a high level interaction model with several limitations about data binding, graphical templates are out of their scope.

**RIA Toolkits**

RIA toolkits, such as JavaFX 2.0 [30], Flex [14], JavaFX 1.3 [24], WPF [22] and Ext GWT [26], provide better data binding capabilities than the two previous categories. Data bindings are often specified as string values within XML files representing the interface (e.g., XXML for Flex or XAML for WPF). An alternative is to use a programming language, such as JavaFX script for JavaFX 1.3. WPF and Ext GWT use their own template model to represent simple graphical components (e.g., items of a list), but this model does not allow the definition of new nesting components. JavaFX 2.0 uses a hand-written representation of the scene that describes the contents of the template. JavaFX 1.3 provides a bind statement for loops on collection that allows the definition of nesting components. As a summary, RIA toolkits support a large range of data binding capabilities but propose poor interaction models and do not support nesting graphical templates.

**Template-based systems**

As far as we know, Hayaku [23] is the only toolkit that specifically tackles the definition of post-WIMP graphical components. Using abstract representations, Hayaku ends up with good expressiveness and good overall performances. However, Hayaku does not specifically focuses on data binding except for linking the template to its data. Hayaku interaction model provides the mandatory picking capabilities. Within the selected tools that are not specific to GUI engineering, incXSLT [27], sXBL [28] and eXAcT [2] all work with a data model based on XML that we consider as a limitation here. incXSLT is an incremental XSLT processor that allows the incremental transformation of an XML document with the limitation that it does not cover the whole XSLT 1.0 language. We can use incXSLT to define graphical templates as standard &lt;template name&gt; XSLT elements. The SVG’s XML Binding Language (sXBL) supports the parametrization of SVG symbols natively.; However, it offers limited data binding capabilities. eXAcT is an incremental/active Java-based transformation processor that offers good data binding capabilities. Loa uses the concept of fragment and anchor as defined by eXAcT while leveraging the inherent complexity of eXAcT transformations.

“Template-based systems” are relevant approaches for the definition of graphical templates. However they provide poor support for the integration of an interaction model and of a data model.

**Data binding languages**

Definitive principles and notations [5] proposes a novel approach for the programmatic evaluation of functions. For instance, instead of calling $y=f(x)$ each time $x$ changes, definitive notations represent $y=f(x)$ as a constraint $f$ that binds $x$ and $y$. This works well for property bindings but there are limitations for collection bindings. Notations are defined within a dedicated language, not close to Object Oriented Programming (OOP) usage. Active operations are based on a similar principle [3] but allows the definition of data bindings using a classic OOP approach and standard operations on collections [29]. [4] demonstrates that active operations allow the definition of complex data bindings that cannot be expressed with RIA toolkit without ad hoc coding. Loa is based on the concept of active operations implemented as a Scala internal DSL. “Data binding languages” propose languages that support simple and complex data bindings. Graphical templates and interaction models are however not in the scope of these techniques.

**Conclusion**

This preliminary evaluation shows that GUI toolkits (i.e., Java-based, interactor-based and RIA) have little to no support for the definition of nesting templates and interactor models, and limited support for data binding.
The integration of relevant approaches that are not dedicated to GUI, such as XSLT templates and active operations, provides new functionalities that overcome these limitations. Loa has been designed for this very purpose: the successful integration of the various concerns of interactive systems in a unique DSL. Next section provides evaluation and discussion on the Loa language.

**EVALUATION**

In this section, we propose to evaluate Loa and its supporting process against the Cognitive Dimensions of Notation proposed by Green [11]. We discuss the dimensions of abstraction, hidden complexity, closeness of mappings, viscosity and progressive evaluation and support our claims with van Deursen’s observations [25] about the benefits of using DSLs.

**Abstraction / Hidden Complexity**

Providing a good level of abstraction is essential for the designer to be productive in the design of new graphical components. Since DSLs are concise in general [16], we consider that Loa proposes a representative set of constructs with precise semantics that allows designers to focus on each specific concern of the domain rather than on the complexity of the implementation or of the development artifacts.

The use of an homogeneous representation (i.e., the Loa DSL) across five of the six steps of the development process for new graphical components is a very valuable asset towards reliability and maintainability [7, 15] of the final application. The Loa DSL is also provided with a concrete syntax as a textual notation. The textual notation allows designers to set the links between the development artifacts (i.e., data model, data binding, graphical templates and interaction model) that was not cover by Hayaku [23].

**Closeness of Mappings**

The second step of the Loa development process allows application designers and graphical designers to agree on a design that is close to the final product. From an initial sketch of the graphical component, the production of the component may be realized in parallel: the graphical designer works towards a final version of the visuals while the application designers takes care of the implementation and integration of the component into the final application.

**Viscosity**

Resistance to changes is a challenging activity in any process of development. While this paper does not focus on the activity of maintaining consistence as changes occurs, every step of the process is based on an homogeneous representation (i.e., Loa). Homogeneity is one solution to get confidence in the consistency of the various development artifacts instead of manipulating multiple representations.

Since Loa is an internal Scala DSL, we benefit from the existing Scala and Java tooling to detect the side effects of changes at design time. As a matter of fact, the manual propagation of changes is limited to the synchronization of the data model with the data binding, and of the data model with the interaction model.

**Progressive Evaluation**

The development process that supports Loa covers the various specifications (i.e., from the graphical design to its implementation) required to build a fully functional graphical component. However, testing a new graphical component does not require for all steps to be completed. For instance, the design of a new graphical component can be tested against its static properties while the behavior is not yet implemented. This allows designers to design components incrementally.

**CONCLUSION**

This paper presents the Loa framework dedicated to the engineering of interactive systems. Switching from implementation to specification, the Loa framework focuses on the definition of specifications that are executable within a final application. The framework integrates previous works on data binding, graphical templates and interactors. The Loa framework provides: 1) a dedicated DSL based on the Scala language; 2) a six-step development process that provides guidance in the use of the framework; and 3) an implementation that currently bridges the Swing toolkit and allows the definition of graphical components in SVG.

Perspectives of the framework target the implementation of bridges to other Java toolkits, such as JGraph and SWT/JFace, as well as bridges to other platforms such as .NET and Web-oriented technologies. Long term perspectives will target the integration of other concerns specific to UI design such as groupware and constraint management.

**REFERENCES**