Spec

A Framework for the Specification and Reuse of UIs and their Models

Benjamin Van Ryseghem
RMoD, Inria Lille – Nord Europe
benjamin.van_ryseghem@inria.fr

Stéphane Ducasse
RMoD, Inria Lille – Nord Europe
stephane.ducasse@inria.fr

Johan Fabry
PLEIAD Lab – Computer Science Department (DCC)
University of Chile
jfabry@dcc.uchile.cl

Abstract

Implementing UIs is often a tedious task. To address this, UI Builders have been proposed to support the description of widgets, their location, and their logic. A missing aspect of UI Builders is however the ability to reuse and compose widget logic. In our experience, this leads to a significant amount of duplication in UI code. To address this issue, we built Spec: a UIBuilder for Pharo with a focus on reuse. With Spec, widget properties are defined declaratively and attached to specific classes known as composable classes. A composable class defines its own widget description as well as the model-widget bridge and widget interaction logic. This paper presents Spec, showing how it enables seamless reuse of widgets and how these can be customized. After presenting Spec and its implementation, we discuss how its use in Pharo 2.0 has cut in half the amount of lines of code of six of its tools, mostly through reuse. This shows that Spec meets its goals of allowing reuse and composition of widget logic.

1. Introduction

Building user interfaces is a notoriously time-consuming task. To help developers in their tasks, several approaches have been proposed previously. The basic principle of decoupling the model from its view: MVC [3] was first proposed in Smalltalk-80. This principle was later evolved to Model View Presenter (MVP) [4] by the Taligent project. In MVP, the presenter assumes part of the functionality of the controller and is the sole responsible for coordinating how the UI manipulates the underlying model. The view is now also responsible for handling UI events, which used to be the controller's responsibility.

Orthogonally to these concepts, UI builders were developed as tools to facilitate the design and building of UIs. The goals of UI builders are often twofold: firstly to support the description of widgets (location, size, color) and their logic, and secondly the composition and reuse of existing component logic. VisualWorks [5, 6] was a pioneer of this approach. Its builder is based on application classes that glue widgets and domain objects together, based on a literal description of widget properties.

Important issues with UI builders stem from the fact that their working is often based on direct code generation from the UIBuilder visual pane. As a first consequence the simple fact of reloading a UI description in the builder for editing, arguably a common occurrence, is already a complicated process. This complication arises because the UI description code has to be interpreted differently from a normal execution, since the normal execution opens the UI for use. Secondly, there is still the challenge of reusing widgets and their interaction logic. In our experience with Pharo, the use of UI builders there has led to a significant amount of code duplication which can be avoided. For example, the Senders/Implementors tool shows a list of methods and displays their source code. Pharo also provides the VersionBrowser, which displays methods as a list and their source code. Furthermore the ProtocolBrowser displays all the methods of a class and their source code. These three tools are mostly duplicated code, essentially implementing three times the same behavior and the same UI, with some superficial differences. In our opinion this code duplication arises because the widgets are not generic enough to be reused while also being able to be adapted to cope with (subtle) variations.

To address the above issues, two underlying design choices need to be taken: (1) how do we define UI descriptions and (2) how do we compose the logic of UIs. We assert that there is a need for a declarative way to specify UIs that also allows for seamless composition and reuse of the UI declaration and logic. In line with this assertion we have de-

IWST '12 August 28th, 2012, Gent, Belgium.

veloped Spec, a UI builder for Pharo, and we present it in this article.

With Spec, widget properties are defined declaratively and attached to specific classes known as composable classes. These composable classes also act as presenters by defining the bridge to the underlying model, in addition to the widget interaction logic. Spec reuse includes these presenters, i.e. Spec allows for the reuse of widget logic and their composition as well as their connection to the underlying model. This support for reuse is not only novel but we also consider it the most important contribution of Spec, as its use removed a high amount of code duplication in Pharo 2.0.

This paper is structured as follows: we next detail the issues that emerge from the current UI builder approach. This is followed, in Section 3, by an introduction of Spec that builds a number of UIs highlighting reuse. Section 4 gives an overview of the more salient points of its implementation and, in Section 5, we provide a more formal description of the different elements used in the example. Spec is currently used in Pharo, and we talk about this in Section 6. Related work is discussed in Section 7, and Section 8 concludes.

To avoid any ambiguities in this text due to issues with terminology, we first define three terms briefly, since these typically have overloaded meanings.

UI Element: an interactive graphical element displayed as part of the Graphical User Interface.

UI Model: an object that contains the state and behavior of one or several UI elements.

Widget: the union of a UI Element and its UI model.

2. UI Builder Challenges

A UI builder is a tool used to generate user interfaces. Such builders help the developers by providing a framework for UI construction on top of the UI libraries provided by the language. They may also provide a UI for graphically building a UI. Put differently, a UI builder is *not necessarily* a tool with a UI although it may provide a UI to interactively place widgets on a canvas.

To be able to help the developers in creating UIs, UI builders usually provide support for:

- the description of widgets (color, size, visual effect, specific behavior,...)
- the description of their placement
- the definition of the widget behavior, *e.g.,* how it reacts to certain events.

However in our experience the above is not enough. This is because developing UIs is not only about widget generation, but also about the reuse of the logic between widgets. As Pharo maintainers we have seen that most of the UIs of the tools present in Pharo were written from scratch. This even if a lot of tools are essentially manipulating the same objects and rendering them more or less the same way. This lack of reuse makes the system harder to maintain and slows down enhancements to the UIs of these tools. To address these problems, UI builders should also support the reuse of widget logic and composition. We have seen that the logic of one widget is often based on the wiring of the logic of adjacent or nested widgets. Hence being able to compose and reuse existing behavior is central to be able to build new widgets.

The goal of reusability however brings a new problem. Indeed, if the widgets must be reusable it means that on one hand the widgets must be generic enough to be used in different scenarios and on the other hand they should be parametrizable enough to fit these new scenarios.

To enable the reuse in the process of building and maintaining the UIs of Pharo, we have built Spec. Spec is a new UI builder whose goal is to support UI interface building and seamless reuse of widgets. Spec is based on two core ideas: first the declarative specification of the visual behavior of widgets and their logic, and second inherent composability of widgets, based on explicit variation points.

Figure 1 shows the principles of Spec: a UI is built from composed widgets that are glued together using ports and whose visual characteristics are defined using a declarative specification that are reused over composition.

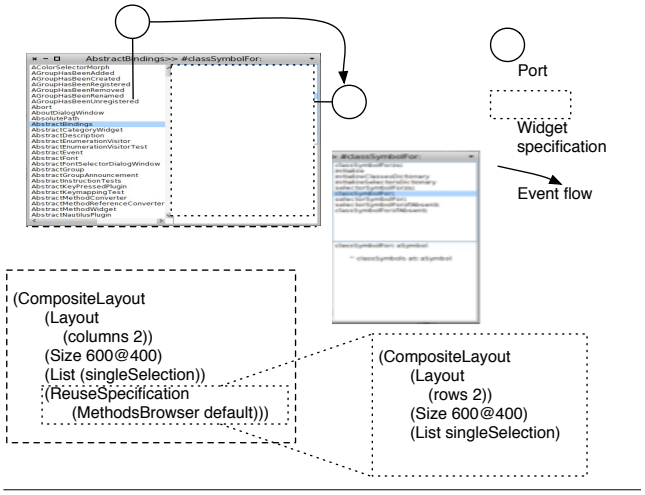

Figure 1. Spec principles

3. Spec by Example

In this section we introduce Spec and its key points by showing an example of the typical use of Spec. Note that we only focus here on the logic of the building process, the discussion of a graphical tool to compose widgets based on Spec is out of the scope of the paper.

The example we develop in this section starts with showing how basic widgets are composed to build a simple UI and continues with illustrating how these composed widgets can be reused and adapted to build more substantial UIs. In total we build three UIs: a Method List that shows a collection of

methods, in Section 3.1, a Method Browser that reuses the list and adds a pane showing the source code of the method, in Section 3.2, and lastly a Class Browser, reusing and adapting the Method Browser, in Section 3.3.

3.1 Methods List

In this section, we present how to build a method list in five steps:

1. the creation of the class;
2. the implementation of the initialize process;
3. the implementation of the getters;
4. the specification of a layout;
5. the window title.

We will now present these five steps in more detail.

Class Creation. First we need to create a class named MethodsList.

```
ComposableModel subclass: #MethodsList
  instanceVariableNames: 'list'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'IWST12-Spec'
```

Here we can see two things:

- the superclass is ComposableModel: this class is the root of the Spec UI model hierarchy.
- the instance variable *list*: an instance variable needs to be defined for holding the UI model that will represent our methods list.

Initialization. Second, the initialization. The initialization of a UI is done in three different methods:

1. initializeWidgets: to set the instance variables which hold sub models and their associated widgets and to configure these sub UI models;
2. initializePresenter: to wire sub UI models together;
3. initialize: to initialize remaining state of the UI.

In this example, only the widget instantiation has to be done in the initializeModels method of the MethodsList class, such that it sets the *list* instance variable to contain a ListComposableModel with its associated list UI element.

```
MethodsList>>initializeWidgets
    self instantiateModels: { #list -> #ListComposableModel }.
```

The code above shows how the model for the list is instantiated, in a declarative fashion. The method instantiateModels: allows one to provide a collection of associations where the key is the instance variable name and the value is a UI model class. Hence the code above creates a new instance of ListComposableModel (and its associated list UI element) and stores it in the instance variable *list*.

Accessors. Third the accessor to the instance variable has to be implemented, such that the Spec infrastructure can obtain this list instance when required.

```
MethodsList>>getList
    ^ list
```

Layout Specification. Fourth we specify a layout: the object that is used to describe and represent the layout of the UI elements. This is done by implementing a method that returns it on the class side of the UI model. For our example, we implement MethodsList class»myFirstSpec on the class side of MethodsList as follows:

```
MethodsList class>>myFirstSpec
    <spec: #default>

    ^ SpecLayout composed
        add: #getList;
        yourself.
```

Since multiple layouts per UI model can be present, there is a mechanism to set the layout to use by default. There are two ways to define the default specification to be used:

1. pragma: all specifications are tagged with a pragma <spec:> allowing the spec infrastructure to correctly retrieve the corresponding method. In addition, the keyword default can be used in the pragma to specify that this layout has to be used by default.
2. method name: if there is no <spec: #default> tag, the method named *defaultSpec* is used.

The code above uses a pragma and simply returns the layout object for this UI. Sending the message composed to the SpecLayout class yields a composed layout, which allows one to compose the different models that are part of Spec. To this composed layout, the add: message is sent, adding the argument to the layout. In this case we provide the symbol getList, the selector of the getter of the model we want to include into the methods list. Note that, in general, this argument may be a SpecLayout as well, allowing for the reuse of high level composed models, as we shall show later.

Executing the following snippet displays the generated widget embedded in a window using the above, default, specification.

```
MethodsList new openWithSpec.
```

To populate the list, the message items: has to be sent to the instance variable list with a collection of items to be shown as argument. In our example this would be a collection of methods, *e.g.,* the methods of the class Object as shown below:

```
(MethodsList new
    openWithSpec;
    yourself)
    getList items: Object methods
```

By default, the method printString is sent to each item to produce the string used to display the item in the list. Often the default behavior displays to much information, or is not accurate enough. To address this, a block can be used to specify how to generate the display string. This is achieved by sending the message displayBlock: to the list widget. The following code provides an example of how to specify a display block in the initializeWidgets method such that the displayed string follows the form class name»selector.

```
MethodsList>>initializeWidgets
    self instantiateModels: { #list -> #ListComposableModel }.
    list displayBlock: [:method || name |
        name := method methodClass name.
        name, '>>#', method selector ].
```

When the window is opened, each item is displayed as shown in Figure 2.

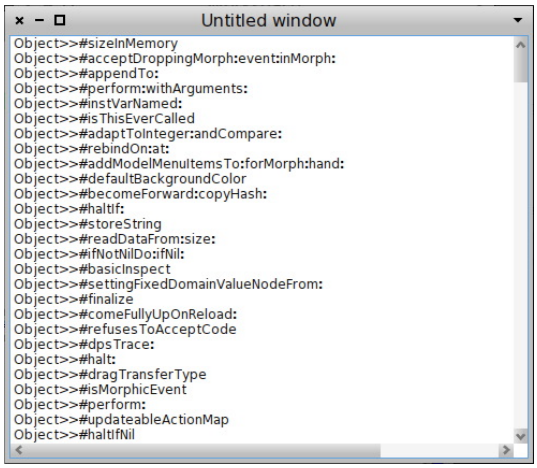

Figure 2. The methods list with a specific display

Window Title. Fifth and last, we show how to change the window title according to the current list selection. By convention, the window title is determined by the return value of the method named title.

```
MethodsList>>title
    ^ list selectedItem
        ifNil: [ 'Methods list' ]
        ifNotNil: [:method | method selector ].
```

The code above returns the selector of the selected method, or the string 'Methods List' if no item is selected.

In order to update the title each time the selected item of the list has changed, we must relate the selection action in the methods list to the updating of the window title. This hence needs to be implemented in the initializePresenter method, and is as follows:

```
MethodsList>>initializePresenter
    list whenSelectedItemChanged: [ self updateTitle ].
```

The code of the method states that the title of the window should be updated when the selected item in the list changes. In Figure 3 we show the result of these changes: a window where the title is the selector of the selected item.

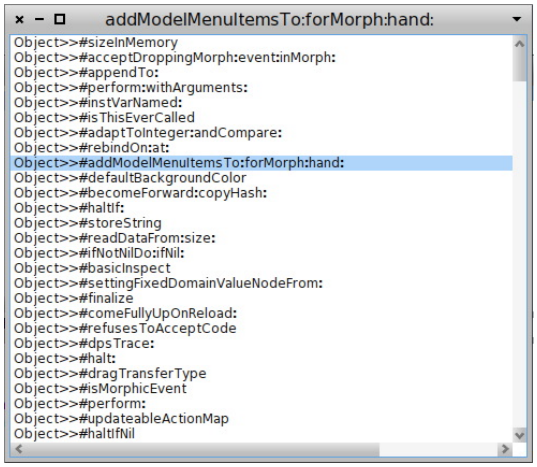

Figure 3. The methods list with a dynamic title

Public API. This completes the construction of the UI model. However if we want this UI model to be reused and embedded we must provide a public API for it. In our example, we want to have the API of MethodsList polymorphic to the core protocol of the embedded list. Hence for each method of the API we forward the message to the *list* instance variable, as shown below:

```
MethodsList>>items: aCollection
    list items: aCollection

MethodsList>>displayBlock: aBlock
    list displayBlock: aBlock

MethodsList>>whenSelectedItemChanged: aBlock
    list whenSelectedItemChanged: aBlock

MethodsList>>resetSelection
    list resetSelection

MethodsList>>selectedItem
    list selectedItem
```

Those methods will be used in the following section when defining its public API, which is used in our last example (in Section 3.3).

3.2 Methods Browser

To show how Spec allows for the reuse of existing models, the next step of our example builds a message browser which reuses the method list we just constructed. We will follow the same five steps as previously:

1. the creation of the class;

2. the implementation of the initialize method;
3. the implementation of the getters;
4. the specification of a layout;
5. the window title.

These steps will now be presented in more details.

Class Creation. First we define a class, this time named MethodsBrowser.

```
ComposableModel subclass: #MethodsBrowser
    instanceVariableNames: 'methodsList text'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'IWST12-Spec'
```

The class has two instance variables:

- *methodsList*: the list for displaying the methods. It will be an instance of the MethodsList class we defined in Section 3.1, i.e., we reuse the methods list we defined previously.
- *text*: the text zone used to display the source code. It will be an instance of TextModel.

Initialization. Second, in the initialize process, we have to instantiate the value for each instance variable.

```
MethodsBrowser>>initializeWidgets
    self instantiateModels:
      {  #methodsList    -> #MethodsList.
         #text           -> #TextModel }
```

The above code shows that a model that is being instantiated can be a standard class of Spec as well as any UI model class that has been defined using Spec. This is the key feature that allows for seamless reuse of models in Spec.

Next, we specify the overall behavior of the UI, linking the two widgets together. When an item from the list is selected, the text area should display its source code.

```
MethodsBrowser>>initializePresenter
    methodsList whenSelectedItemChanged: [:method |
        text text: (method
                ifNil: [ '' ]
                ifNotNil: [ method sourceCode ]) ]
```

The initializePresenter method above specifies the overall behavior of the UI. It links the list to the text field by stating that when a item in the list is selected the text of the text zone is set to:

- an empty string if no item is selected
- the source code of the selected item otherwise.

Accessors. Third we implement the getters needed by the layout.

```
MethodsBrowser>>methodsList
    ^ methodsList

MethodsBrowser>>text
    ^ text
```

Layout Specification. Fourth we specify a presentation, by defining a method at the class side.

```
MethodsBrowser class>>spec
    <spec: #default>
    ^ SpecLayout composed
        add: #methodsList origin: 0@0 corner: 1@0.5;
        add: #text origin: 0@0.5 corner: 1@1;
        yourself
```

In addition of showing that the reuse of a Spec model is transparent with regard to the layout, the above method also shows that a position can be specified for each sub-model. Here, methodsList will be displayed in the top-most half of the generated widget while text will be displayed in the bottom-most half of the generated widget.

Since most of the UI elements can be expressed in terms of rows and columns, a simpler way to describe UI elements is also available. The following code produces the exact same layout as previously, in a more concise and more readable way.

```
MethodsBrowser class>>spec
    <spec: #default>
    ^ SpecLayout composed
      newColumn: [:c |
          c
            add: #methodsList;
            add: #text ];
      yourself
```

The following snippet opens the widget and populates the list with the methods of Object:

```
(MethodsBrowser new
    openWithSpec;
    yourself)
    methodsList items: Object methods
```

Figure 4 shows the result of executing the above code.

Window Title. When an item is selected the title is not updated as it used to be in MethodsList. This is addressed by implementing a title method, and slightly modifying the initializePresenter method as follows:

```
MethodsBrowser>>title
    ^ methodsList title


MethodsBrowser>>initializePresenter
    methodsList whenSelectedItemChanged: [:method |
        self updateTitle.
        text text: (method
                ifNil: [ '' ]
                ifNotNil: [ method sourceCode ]) ]
```

This gives us a dynamic title, as shown in Figure 5.

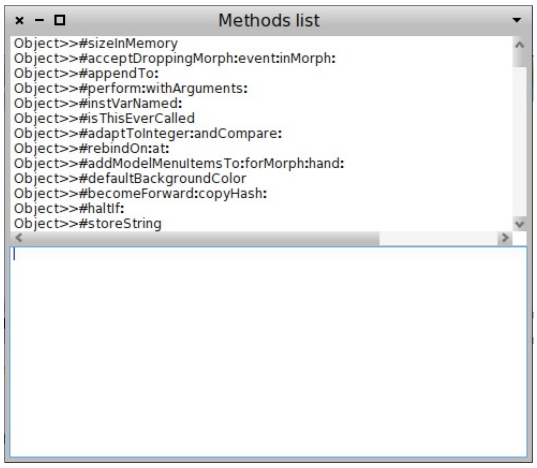

Figure 4. The methods browser

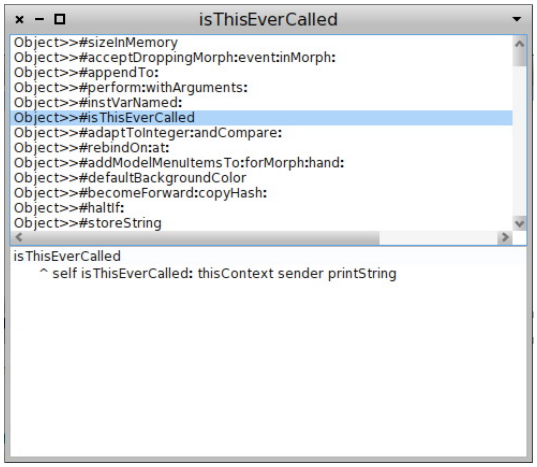

Figure 5. The methods browser with the dynamic title back

Public API. Recall that to have a reusable model, we need to define its public API. We will reuse this model in our last example, therefore below we show the methods of the public API for our MethodsBrowser. As these are straightforward we do not discuss them in more detail.

```
MethodsBrowser>>items: aCollection
    methodsList items: aCollection

MethodsBrowser>>displayBlock: aBlock
    methodsList displayBlock: aBlock

MethodsBrowser>>resetSelection
    methodsList resetSelection

MethodsBrowser>>selectedItem
    methodsList selectedItem
```

Note that the methods we invoke here are part of the public API we defined at the end of section 3.1.

Conclusion. This concludes the construction of our methods browser. In the construction of this browser we have shown how we can straightforwardly reuse existing models and how a selection in one widget can be used to update the display in another widget, effectively linking widgets together.

3.3 Classes Browser

The last example we provide shows how to parametrize the reuse of models, and how a UI communicates with the underlying model. We do this by illustrating how to build a simple class browser that reuses the MethodsBrowser. Being able to deeply parametrize models allows for extended reuse possibilities of the models since they can be more generic.

The five construction steps essentially are still the same, therefore we first only show a overview of the browser construction process. We will instead focus on how the reuse of user interface specifications can be parametrized. Second we modify the behavior of the reused UI model, and third modify the layout of the reused models. Fourth and last we show how to connect the browser to the class structure, i.e. to its model.

Basic Reuse. The class for the browser is called ClassesBrowser and it has two instance variables: *classes* for the list of classes, and *methodsBrowser* for the list of methods and the text zone, i.e. the methods browser we constructed above. The layout specification below shows how the two are laid out.

```
ClassesBrowser class>>defaultSpec
    <spec>
    ^ SpecLayout composed
          newRow: [:r |
             r
                add: #classes;
                add: #methodsBrowser ];
       yourself.
```

For brevity, we omit the methods that set the title, the accessors and the initialization methods that instantiate the widgets and link them together. The below snippet pops up the window shown in Figure 6 and populates it with all the classes of the system.

```
(ClassesBrowser new
       openWithSpec;
       yourself)
       classes items: Smalltalk allClasses
```

UI Parametrization. We now present how to parametrize the behavior of reused UI models. When reusing a UI model, the reusing UI model can override all parametrization parameters, *e.g.,* the displayBlock, of the reused UI model. This is done by simply providing a new parameter for the reused UI model. It overrides any parametrization made inside of the reused UI model.

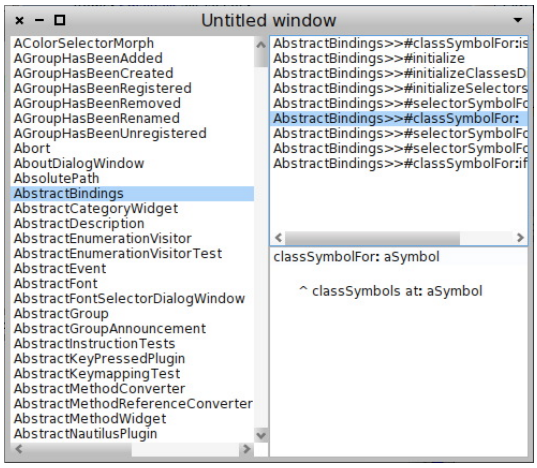

Figure 6. Classes browser

In our example, when a class is selected in the classes browser, its methods are displayed. But they are displayed following the form class name»selector, where the class name is redundant as it is already given by the list selection. Also, the title shown is not correct: it displays the default title, 'Untitled window', even when items in the method list are selected. To perform these two parametrizations, we modify the initialization methods of ClassesBrowser to override the behavior specified in MethodsBrowser. The modifications consist in adding one line to the method, as shown below.

```
ClassesBrowser>>initializeWidgets

"... omitting code for behavior shown previously ..."
    methodsBrowser displayBlock: [:method | method selector ].

ClassesBrowser>>initializePresenter

"... omitting code for behavior shown previously ..."
   methodsBrowser whenSelectedItemChanged: [ self updateTi-
tle ].
```

Recall that these methods we are calling are part of the API we defined at the end of Section 3.2.

The default behavior of Spec is to ignore the window title logic of reused models, hence the presence of the default title. The last line above configures the reused method browser to update the title on a list (de)selection, using the title method defined in the classes browser. We therefore also have to implement ClassesBrowser»title. The following implementation inspects the selection of method browser to return the appropriate value. It returns 'Classes Browser' if nothing is selected, or the selected class name if only a class is selected, or class name »selector if both a class and a method are selected.

```
ClassesBrowser>>title

    ^ classes selectedItem
        ifNil: [ 'Classes Browser' ]
        ifNotNil: [:class | methodsBrowser selectedItem
                    ifNil: [ class name ]
                    ifNotNil: [:method | class name, '>> #', method se-
lector ]].
```

As shown in Figure 7, the methods are now displayed using only their selector and the title follows the form class name »selector, since a method is selected.

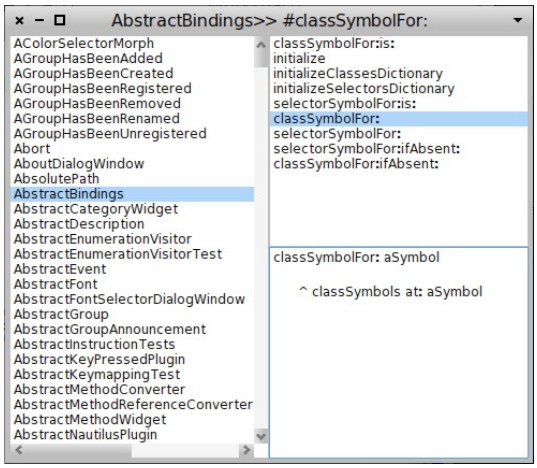

Figure 7. Classes browser with methods displayed using their selector

Layout Specification. The current layout of the ClassesBrowser view is somewhat unusual. The classic layout for such a tool is to have the two lists in a row in the top-most half and the text zone in the bottom-most half. To do this, a new layout is provided by implementing a new spec method, and set as the default spec:

```
ClassesBrowser class >>moreClassicalSpec
   <spec: #default>

   ^ SpecLayout composed
       newColumn: [:c |
         c
           newRow: [:r |
             r
                 add: #classes;
                 add: #(methodsBrowser methodsList) ];
            add: #(methodsBrowser text) ];
        yourself
```

In the above code, instead of using the *methodsBrowser* layout we define precisely how we want the sub-widgets to be rendered.

The result is the widget shown in Figure 8: a classes browser with the two lists in the upper part and the text zone in the lower part.

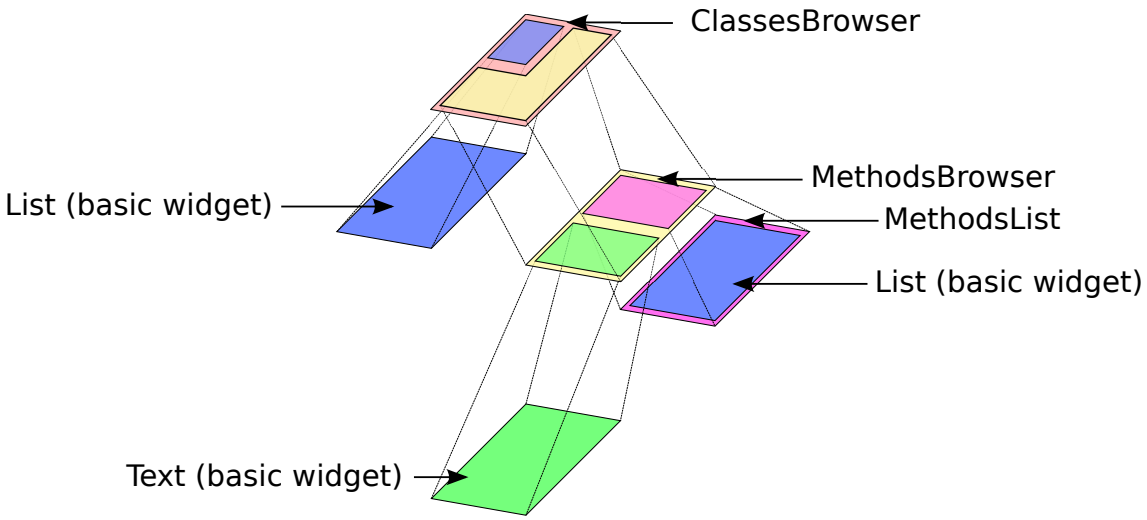

Figure 9. The three levels of layers composing the ClassesBrowser UI

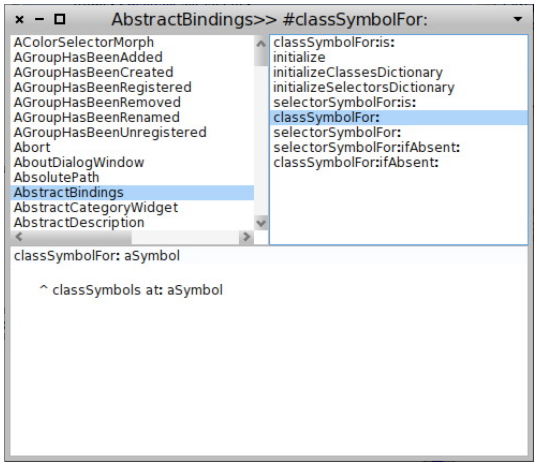

Figure 8. Classes browser with a classic layout

Figure 9 exposes a view of the different layers comprising the ClassesBrowser UI.

Connecting the UI to the Underlying Model. As is, the UI we built is disconnected from the model it is representing: after we pass it an initial list of classes we are not able to modify it, *e.g.,* by editing methods, and it is not aware of any modifications made to its model by other browsers. We now show how to connect the UI to the underlying model, by specifying menu actions to enable the former, and subscribing to announcements to implement the latter. As this behavior belongs in the methods browser, we specify it there, and it will automatically also be present in the classes browser.

The text widget present in the methods browser provides for a standard code editing menu, where we can specify the action to take whenever the code is saved or accepted. This is performed by providing an accept block, as illustrated below:

```
MethodsBrowser>>initializePresenter
  "... omitting code for behavior shown previously ..."
   self text acceptBlock: [ :text :notifier | ... compile the text ... ].
```

The code obtains the text field from the methods browser and specified that the code, contained in the text parameter should be compiled. (We do not include the compilation code as it is not pertinent to this discussion.)

Lastly, to have the browser to react to changes in the underlying class structure, we use the system announcements mechanism to refresh its contents when needed. We rely on the fact that when such changes occur, the system raises an announcement, and subscribe to these announcements, for example using the code below the browser becomes aware of methods being added, adding them to the list of methods shown.

```
MethodsBrowser>>initializePresenter
   "... omitting code for behavior shown previously ..."
   SystemAnnouncer announcer weak
      on: MethodAdded send: #methodAdded: to: self.

MethodsBrowser>>methodAdded: anAnnouncement
   | sel text it |
   text := self text pendingText.
   sel := self methodsList selectedItem.
   it := anAnnouncement item.
   self items: (self methodsList: listItems add: it; yourself).
   self methodsList setSelectedItem: sel.
   self textModel pendingText: text.
```

The methodAdded: method first keeps a copy of the text being shown in the text field, as it may contain edits that have not been saved. It then obtains the selected item in the

methods list, and adds the new method to the list. As the change in list items may change the selection in the list, it then sets the selected item to the previously stored value. Lastly, it sets the text being shown in the text field to the value stored in the first step.

Conclusion. This concludes the construction of our last example: a methods browser. In this section we have seen how it is possible to parametrize both the behavior and layout of UI models that are being reused. This allows for more generic models to be reused and customized further when needed, increasing their reuse possibility. We have also shown how to connect the user interface to an underlying model, yielding a fully functional UI.

4. The Implementation of Spec

The implementation of Spec is based on two pillars: the presence of a SpecLayout and the use of value holders. The SpecLayout is used to describe how the graphical elements are positioned inside the generated UI, by using either basic widgets or by reusing composed widgets. The value holders are used to store the variation points of the model and to react precisely to one of these changes. This way the UI updates are less frequent and more precise, and hence faster.

In this section we will discuss these two points in more detail, firstly talking about SpecLayout and secondly discussing the value holders.

4.1 SpecLayout and its Interpreter

The SpecLayout describes and represents the layout of the UI elements in Spec. More details about how to manipulate SpecLayout objets will be provided in Section 3.1. A Spec interpreter is used to build a UI widget from this layout. To have a static structure which can be easily used by the interpreter, the SpecLayout object is converted to an array before it is passed to the interpreter.

We have only seen here the composed layout, consisting of an aggregation of different Spec widgets. However there are additional kinds of layouts provided: one kind for each type of widget. This allows one to attach specific behavior to each spec type, *e.g.,* the class of the UI element specific to this kind of layout. The array representation of a spec has as first element an identifier of its type.

By using a spec type, default behavior can be defined and shared among all the widgets. It means that all the widgets produced by a defined spec type can be changed by modifying the spec type itself. Types also allow the creation of a data structure representing the tree of sub-widgets and to use a visitor pattern to build tools on top of specs, *e.g.,* like a UI builder.

The remainder of the array representation of a spec can be seen as a stack: each time a selector is read, as many arguments as needed by the selector are popped and analyzed. The spec interpreter iterates over a SpecLayout array and builds each part of the widget by recursively interpreting each element.

A SpecLayout allows one to specify where to position a widget's sub-widgets, and also allows one to position the sub-widgets of sub-widgets. This provides a way to reuse generic widgets while being able to customize them sufficiently to make them conform to a new usage scenario.

4.2 Value Holder

A value holder is a simple object which wraps a value and raises an announcement when this value is changed. Thanks to this mechanism we can use value holders as wrappers around model values and make the UI react at specific changes to the model. As such, we provide an event based structure that allows one to react to only to value change of interest.

The above means that, for example, the selection index of a list is stored in a value holder. When a new item is selected, the index changes, and the corresponding value holder raises an announcement. The basic widgets of Spec provide as part of their API event registration methods, which allow a user-defined object to react to this change if needed.

Moreover, having a value holder for each model data allows one to update the UI only for the data which has changed, without having to examine this change to establish its relevance as the DependencyTransformer in VisualWorks. This is in contrast to classical MVC [3] and MVP [4]. Here, when the observable object has changed, it sends an `update:` message to all observer objects with the changed value as argument. Then in the `update:` method, the observer has to examine the argument to react in accordance with the change. In Spec, the observer registers to each observables' value holder it is interested in, and for each value holder specifies a method to invoke when the value holder is changed. Hence the examination of the updated value is no longer necessary and the dispatch to the appropriate update logic is done naturally without any switch case.

In addition, since the whole event flow is controlled and propagated through value holders, Spec does ensure that there are no event loops due to circular event sends.

Note that since every object can register to a value holder changes, this means that a model can register itself to any of it sub-widgets value holders, or any sub-widgets sub-widgets value holder. Thanks to this, a model can add new behavior for its sub-widgets. This provides a way to reuse generic widgets while being able to parametrize them enough to make them correspond to a new scenario.

5. The spec of Spec

In this section we summarize the specification of the public APIs of the relevant building blocks for a user of Spec: the basic widgets and `SpecLayout`.

Selector	Result
displayBlock:	set the block used to produce the string for displaying items
items:	set the contents of the list
resetSelection	unselect selected items
selectedItem	return the last selected item
whenSelectedItemChanged:	set the block performed when the selected item changed

Table 1. ListComposableModel public API

Selector	Result
accept	force the text zone to accept the pending text
acceptBlock:	set the block to perform when the pending text is accepted (saved)
text:	set the text of the widget to the value provided as argument
whenTextIsAccepted:	set a block to perform when the text is accepted
whenTextChanged:	set a block to perform when the text has changed

Table 2. TextModel public API

5.1 Models public API

To build a UI the user combines basic UI models and existing Spec models as required. For Spec there is however no distinction between these two, as basic UI models are reified as Spec models. Put differently, these basic UI models are Spec models that simply wrap the widgets that are provided by the GUI framework.

Due to lack of space, we do not provide a complete specification of the public API of all models provided by Spec (11 models, in total 228 methods). The complete API for all models is provided as part of a tech report about Spec [14]. We restrict ourselves here to the public API methods of the basic models used in this paper: ListComposableModel, shown in Table 1, and TextModel, shown in Table 2.

5.2 SpecLayout

A SpecLayout is an object used to describe the layout of the UI elements of a widget.

The SpecLayout class provides a small API (only 10 methods), shown in table 3. The *add* methods and the *newRow* and *newColumn* methods cover the bulk of the use cases: adding elements to the layout. Indeed, as we have seen in Section 3 they are the only methods used when the layout is a composed layout.

The remaining two *send* methods are required to be able to interact with basic widgets. Since the Spec reification of basic UI models provides a bridge between Spec and a graphical library, the class of the UI element nor its API can be predicted. Hence we need to be able to send any message to those classes through the SpecLayout. To allow for this, the SpecLayout provides for the *send* methods, which enable performing any selector with the corresponding arguments. Thanks to these methods we ensure that a bridge can be built between Spec and any graphical library.

As an example use of the send:withArguments: method, the following code is the implementation of TextModel class»defaultSpec, which defines the binding between Spec and the Morphic UI framework for the TextModel widget. (Due to the low-level nature of this code we do not explain its functionality in detail.)

```
defaultSpec
    <spec>
    ^ SpecLayout text
            send: #on:text:accept:readSelection:menu:
            withArguments: #(model getText accept:notifying: read-
Selection codePaneMenu:shifted:);
        send: #classOrMetaClass: withArguments: #(model behavior);
            send: #enabled: withArguments: #(model enabled);
                send: #eventHandler: withArguments: #(Even-
tHandler on:send:to: keyStroke keyStroke:fromMorph: model);
            send: #vSpaceFill;
            send: #hSpaceFill;
            yourself
```

6. Spec in Pharo

Spec has been introduced in Pharo 2.0 with the goal to be used for rewriting all the tools. For now, six different widgets have been implemented:

1. MessageBrowser: a tool made for browsing messages (similar to the MethodsBrowser made in section 3.2);
2. Senders/Implementers: a tool to browse the senders or the implementors of a given selector;
3. ProtocolBrowser: a tool to browse all the methods that a given class can understand;
4. VersionBrowser: a tool used to browse the different versions of a provided method;

Selector	Result
add:	add the object provided as argument. This object can be the selector of a getter to an instance variable storing a ComposableModel or another layout.
add:origin:corner:	add the object provided as argument and specify its position as fractions.
add:origin:corner:offsetOrigin:offsetCorner:	add the object provided as argument and specify its position as fractions and offsets.
add:withSpec:	add the model provided as first argument using the selector provided as second argument for retrieving the spec. The first argument can be the selector of a method that returns a ComposableModel or a collection of such selectors.
add:withSpec:origin:corner:	add the model provided as first argument using the selector provided as second argument for retrieving the spec. and specify its position as fractions.
add:withSpec:origin:corner:offsetOrigin:offsetCorner:	add the model provided as first argument using the selector provided as second argument for retrieving the spec. and specify its position as fractions and offsets.
newColumn:	add to the current layout a column created using the block provided as argument
newRow:	add to the current layout a row created using the block provided as argument
send:	send the message with selector specified as first argument to the underlying widget
send:withArguments:	send the message with selector specified as first argument and arguments specified as second argument to the underlying widget.

Table 3. SpecLayout public API

Tool	Pharo 1.4	Pharo 2.0	Percentage of reduction
MessageBrowser	MessageSet 488	MessageBrowser 329	33%
Senders/Implementers	FlatMessageListBrowser 463	MessageBrowser + 4	99%
ProtocolBrowser	ProtocolBrowser 49	MessageBrowser + 20	47%
VersionBrowser	VersionsBrowser 312	NewVersionBrowser 57	82%
ChangeSorter	ChangeSorter (970) + DualChangeSorter (39) 1009	ChangeSorterApplication (410) + DualChangeSorterApplication (186) 596	41%
Total	2321	1006	57%

Table 4. Comparison between tools in Pharo 1.4 and Pharo 2.0

5. ChangeSorter: a tool made for managing the changes of the system;
6. DualChangeSorter: a second tool for managing changes, with focus on the transfer from one change sorter to another.

As a testament to the possibilities of reuse the MessageBrowser is used for the Senders/Implementors, and the ProtocolBrowser. Moreover the DualChangeSorter is made of two ChangeSorter linked together and specialized to add functionality involving the interactions between the two change sorters.

Table 4 shows the difference in the number of lines of code (LOC) used to implement those tools before the use of Spec (Pharo 1.4) and after (Pharo 2.0). The purpose of this table is to emphasize the reduction of code duplication. The table follows the form:

- in the first column the name of the tool which is being compared;
- in the second column the name of the class used to implement this tool in Pharo 1.4 and the number of LOC used to implement it;
- in the third column the name of the class used to implement this tool in Pharo 2.0 and the number of LOC used to implement it;
- the ratio in LOC reduction.

We will now explain the difference for each line in more details.

MessageBrowser. MessageSet is used in Pharo 1.4 to browse a collection of method references. MessageBrowser from Pharo 2.0 covers all the functionalities of MessageSet and even add new features like a topological sort or a update mechanism and the support for methods in addition of method references. Yet MessageBrowser is still smaller because thanks to widget reuse, some data of the UI itself is managed by widgets that are being reused, *e.g.,* index selection management.

Senders/Implementers. FlatMessageListBrowser is used in Pharo 1.4 to browse the senders or implementers of a selector. In Pharo 2.0 we have decided to reuse MessageBrowser since senders and implementers are also a collection of method references. MessageBrowser already covers all the FlatMessageListBrowser functionalities, and moreover adds the topological sort and the update mechanism as well. Only a trivial modification needed to be made to MessageBrowser. Hence the Senders/Implementers browser is actually a MessageBrowser, where we implemented the required API to open the corresponding list of messages. This explains why the number of line for this tool in Pharo 2.0 is so small.

ProtocolBrowser. ProtocolBrowser is used in Pharo 1.4 to browse all the methods that the provided class can understand. Again, MessageBrowser covers all the features of ProtocolBrowser and still adds the topological sort and the update mechanism. As above, MessageBrowser is reused, by adding the logic specific to the ProtocolBrowser. These 20 LOC are the algorithm to collect the relevant methods.

VersionBrowser. NewVersionBrowser provides a new tool in Pharo 2.0 that covers all the functionality of the previous tool. Implemented as its own class, it reuses MessageBrowser for the UI and beyond that only contains version retrieval methods and UI specialization methods. This leads to a low number of LOC.

ChangeSorter. The two tools have been grouped since the implementation in Pharo 1.4 moved the logic of the DualChangeSorter into the ChangeSorter class. ChangeSorter instances are aware of belonging to a DualChangeSorter or not and act accordingly.

In Pharo 2.0 the ChangeSorterApplication class is smaller than the ChangeSorter class because it only knows about itself. It doesn't contain any information about being part of a DualChangeSorterApplication or not. This is because the DualChangeSorterApplication class knows how to reuse ChangeSorterApplication and what logic to add, and as a result is bigger than the DualChangeSorter class.

But when summing up both applications, the Spec implementation is smaller even while covering all the original functionalities. This is for two reasons: firstly because checking ubiquitously for being part of a dual change sorter is expensive in term of lines of code. Secondly for the same reason than for the use of MessageBrowser, relocating UI element management to a sub-widget allows the reusing code to be concise.

Conclusion. In this section we have seen how the reuse provided by Spec is used in Pharo and how this reuse can reduced the number of lines of code (and the code duplication) by almost half. This confirms our assertion that there is a need for a declarative way to specify UIs that also allows for seamless composition and reuse of the UI declaration and logic. Moreover this shows that Spec is an effective tool to address this need. As a consequence of this observation, rewriting all the tools using Spec is a goal for the next version of Pharo.

7. Related Work

Spec is inspired by the VisualWorks [15][16] UI framework, and like it is based on static specifications *i.e.,* the SpecLayout instances at class side. In VisualWorks all the specifications are performed in terms of low level widgets which means that no composed widget can be reused. In contrast, Spec allows the reuse of high level widgets and as a result, the specifications are simpler. Thanks to this fact the UIs can be composed of smaller widgets that make the system more modular and easier to maintain.

Spec also follows the lead of Glamour [17] in favoring an event-based flow through the widgets. However, Spec can be used for every kinds of UI while Glamour is restricted to browsers. Spec widgets also explicitly declare a public API instead of heavily relying on blocks, as Glamour does.

For the UI generation part, Spec is different from tools like NetBeans [13] or WindowBuilder [12] in the sense that they both only provide graphical tools for generating user interfaces while Spec is based on a text based description of the UI. Furthermore, where NetBeans and WindowBuilder generate java code, Spec uses an object and relies on this object for describing the user interface. Instead NetBeans or WindowBuilder use an XML file or parse Java source code.

The disadvantage of this is that if the XML file is edited by hand or if some parts of the generated Java code is refactored these tools are not always able to handle these changes.

In addition of the UI code Spec also provides a framework for the model behavior when NetBeans or WindowBuilder only provide UI elements generation source code. Indeed, Spec can be used to define (and reuse) the logic links between widgets where NetBeans or WindowBuilder can only be used to generate UI elements.

XUL [18] is an XML based language used for describing and reuse widgets through *overlays*. While a group of widgets can be reused, unlike Spec XUL doesn't allow for locally changing the inner logical links. SWUL [19] is a DSL based on the strategy transformation framework that proposes a more declarative syntax for expressing widgets description in Swing. SWUL behaves like XUL in the sense of not being able to locally redefine the behavior of a subwidget.

8. Conclusion

In this paper we presented Spec, a UI builder whose goal is to support UI interface building and seamless reuse of widgets. Spec is based on two core ideas: first the declarative specification of the visual behavior of widgets and their logic, and second inherent composability of widgets, based on explicit variation points.

In our experience maintaining Pharo, we have seen that there is a nontrivial amount of code duplication in UI code which can be avoided and that the logic of one widget is often based on the wiring of the logic of adjacent or nested widgets. Hence being able to compose and reuse existing behavior is central to be able to build new widgets.

We have shown how Spec can be used, by providing three example UIs that highlight the reuse and parametrization features of Spec. This was followed by a more formal specification of the APIs used in the example and an overview of the most relevant points of the implementation. We then showed how Spec enabled a 57% of code reduction in the reimplementation of six UIs of Pharo, thanks to a high amount of reuse of widgets.

The latter shows that Spec provides ample support for reuse of widgets and is an appropriate tool to address the problem of code duplication in UI code. As a consequence it will be the standard UI builder for Pharo 2.0 and all UI tools in Pharo will be rewritten using Spec.

Availability

Spec is part as standard of Pharo 2.0 and is also available in Pharo 1.4, its Metacello configuration is called ConfigurationOfSpec and is available from SqueakSource3 (http://ss3.gemstone.com/).

Acknowledgments

This work was performed in the context of the INRIA Associated Team PLOMO (2012).

References

[1] B. Van Ryseghem, S. Ducasse, J. Fabry, Spec. A framework for the specification and reuse of UIs and their models, in: Proceedings of the International Workshop on Smalltalk Technologies, IWST '12, 2012, to Appear.

[2] T. Reenskaug, MODELS - VIEWS - CONTROLLERS, Tech. rep., Xerox PARC (1979).

[3] G. E. Krasner, S. T. Pope, A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80, Journal of Object-Oriented Programming 1 (3) (1988) 26–49.

[4] M. Potel, MVP: Model-View-Presenter. the Taligent programming model for C++ and Java, Tech. rep., Taligent, Inc., available at: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf (1996).

[5] Parcplace systems, Objectworks Reference Guide, Smalltalk-80, version 2.5, chapter 36, parcPlace Systems (1989).

[6] T. Howard, The Smalltalk Developer's Guide to VisualWorks, SIGS Books, 1995.

[7] T. Reenskaug, The model-view-controller (mvc) its past and present (2003).
URL http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/HM1A93.html

[8] D. Greer, Interactive application architecture patterns, http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/ (August 2007).

[9] M. Alles, D. Crosby, C. Erickson, B. Harleton, M. Marsiglia, G. Pattison, C. Stienstra, Presenter first: Organizing complex gui applications for test-driven development, in: Proceedings of the conference on AGILE 2006, AGILE '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 276–288. doi:10.1109/AGILE.2006.43.
URL http://dx.doi.org/10.1109/AGILE.2006.43

[10] DolphinSmalltalk, Doplhin Smalltalk, http://www.object-arts.com/DolphinSmalltalk.htm (Sep. 2003).

[11] S. R. Alpert, K. Brown, B. Woolf, The Design Patterns Smalltalk Companion, Addison Wesley, 1998.

[12] Eclipse Technology, Windowbuilder user guide, Tech. rep., Google, available at: http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.wb.doc.user%2Fhtml%2Findex.html (2011).

[13] NetBeans, Netbeans IDE, http://www.netbeans.org, archived at http://www.webcitation.org/5p1qB6hNt (2010).
URL http://www.netbeans.org

[14] B. Van Ryseghem, Spec – Technical Report, Tech. rep., Inria – Lille Nord Europe - RMoD, available at: http://hal.inria.fr/docs/00/70/80/67/PDF/SpecTechReport.pdf (2012).

[15] ParcPlace-Digitalk, VisualWorks cookbook, available at: http://www.esug.org/data/Old/vw-tutorials/vw25/cb25.pdf (October 1995).

[16] I. Tomek, The Joy of Smalltalk, available at: http://stephane.ducasse.free.fr/FreeBooks/Joy/6.pdf (September 2000).

[17] P. Bunge, Scripting Browsers with Glamour, Master's thesis, Fakultät der Universität Bern, available at: http://scg.unibe.ch/archive/masters/Bung09a.pdf (April 2009).

[18] Mozilla Developer Network, XUL - MDN, available at https://developer.mozilla.org/en/XUL (2012).

[19] R. de Groot, Implementation of the Java-Swul language: a domain-specific language for the SWING API embedded in Java, Master's thesis, Faculty of Science, Utrecht University, available at: http://strategoxt.org/pub/Stratego/Java-Swul/swul-article.pdf (January 2005).

[20] Microsoft, Asp.net 4.0 official site, www.asp.net.