Seamless Composition and Reuse of Customizable User Interfaces with Spec[☆]

Benjamin Van Ryseghem

RMoD, Inria Lille – Nord Europe

Stéphane Ducasse

RMoD, Inria Lille – Nord Europe

Johan Fabry

PLEIAD Lab – Computer Science Department (DCC)
University of Chile

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 this task, several approaches have been proposed previously. The basic principle of decoupling the model from its view: MVC [**?** **?**] was first proposed in Smalltalk-80. This principle was later evolved to Model View Presenter (MVP) [**?**] 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 [**?** **?**] 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.

A first important issue with UI builders stems from the fact that their working is often based on direct code generation from the visual tool. As a 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

[☆]This paper is an extended version of the IWST'12 paper: "Spec: A Framework for the Specification and Reuse of UIs and their Models"[**?**]

Email addresses: benjamin.vanryseghem@gmail.com (Benjamin Van Ryseghem), stephane.ducasse@inria.fr (Stéphane Ducasse), jfabry@dcc.uchile.cl (Johan Fabry)

UI for use. Secondly, we have 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 issues of readability of UI descriptions and the low amount of reuse of these descriptions, 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 developed Spec, a UI builder for Pharo, and we present it in this article. This paper is an extended version of the IWST'12 paper: "Spec: A Framework for the Specification and Reuse of UIs and their Models"[**?**]. We nearly rewrote the complete article to describe the problems, motivations and solution at a conceptual level.

This paper is structured as follows: we next provide some context and detail the issues that emerge from the current UI builder approach. Then, Section 3 gives a general overview of Spec. This is followed, in Section 4, by an introduction of Spec that builds a number of UIs highlighting reuse. The more dynamic features of Spec are shown in Section 5. Section 6 gives an overview of the more salient points of its implementation and, in Section 7, we provide a more formal description of the API. Spec is currently used in Pharo, and we talk about this in Section 8. Related work is discussed in Section 9, and Section 10 concludes.

2. UI Builder Challenges

We now provide a bit of context on relevant design patterns for the building of user interfaces before going more in depth on the important challenges when building a UI.

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.

2.1. Context

The first motivation for the MVC pattern was to bridge the gap between the human user's mental model and the digital model in the computer [**?**]. MVC was also designed with a reusability challenge in mind: the goal was to be able to compose small views to build more complex views, and to be able to bind several views of several view kinds with the same data model. With MVC the main UI element is the view and a part of the business logic is implemented directly into views and controllers that are tightly bound to views. However this makes programming views time-consuming, and in certain situations, the reusing of views can be difficult. This is because the complexity of building the view tempts developers to allow the model to access the view directly [**?**], binding them directly to the view.

A second problem regarding the MVC pattern is the testability issue. As the domain model is implemented as a separate component and domain model classes are not directly bound to any view, it remains testable. However, testing the view is still a challenge. View implementations can be very complex with the view elements themselves tightly bound to controllers and possibly complex dependencies between view elements [**?**]. Moreover, view elements are made to be interactively manipulated and not triggered by a testing framework.

The MVP pattern [**?**], was developed to address the above issues of view specification complexity, lack of separation of concerns, and UI testability. MVP introduces a mediator between the domain model and the views, called the *Presenter*. Moreover, the controller is split in two parts: first, the native widgets of current UI Frameworks, which

directly manage user interactions and second, the remaining part of the UI logic and the dependencies between visual representations is now implemented within presenters. Several versions of the MVP pattern have been described [**?**], they mainly differ on the way the model, the view and the presenter are bound together and on how they communicate with each other. Implementing a presenter can still be a time-consuming task. However, as a mediator between the data model and the view components, the same presenter can be reused for different data models and for different views. Moreover, the introduction of the presenter can ease the testing of UIs [**?**].

As an example, a straightforward version of the MVP pattern is used in Dolphin Smalltalk [**?**]. In Dolphin, a domain model manages data and implements the application functionalities, and is defined as a subclass of the abstract Model class. A domain model can comprise other domain models or basic value models for simple data types such as String, Text or OrderedCollection. A presenter is implemented as a class which ultimately inherits from the Presenter class. Specific presenter subclasses are included, such as Dialog for modal windows. A particular presenter includes instance variables to hold separate sub-presenters, each responsible for the editing of a part of the domain model. Sub-presenters are wired together using events. The communication between presenters and the domain model is implemented by an Observer pattern within the model and the presenter class [**?**]. Finally, the views, their layout and their decorations are described as external resources. These are designed separately and interactively with the help of a View Composer tool. The view resource and the corresponding presenter parts are bound together by their respective names. For example, if a text field has to be bound to a String presenter, then the name of the text field resource must match exactly the one of the presenter. When the view design is finished, the view resource is saved as a class method of the presenter. Since the MVP pattern allows for separation of concerns between the controller logic and the ways the information is displayed, the data models and the presenters can now be reused independently. However, the problem of reuse of the view and of testing of the view still remain. Regarding the specification of the view, current solutions mainly rely either on dedicated drawing tools or on XML/CSS-like solutions. As a result, reusing a view from one application to another is still very difficult. This point is addressed further in the next section.

2.2. Challenges

When building a UI, a programmer may decide not to use a dedicated tool and directly create and assemble widgets. This practice however often does not lead to a clean separation between the UI model and its presentation counterpart. In addition, depending on the frameworks used, such practice does not promote further reuse based on composition of existing or produced UI elements. Consequently, using adequate tooling, *i.e.,* an UI builder, is advisable.

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 UI libraries composed of widgets. They may also provide a UI for graphically building a UI but this is optional. A UI builder is however *not necessarily* a tool with a UI. It may elect to provide a UI to interactively place widgets on a canvas, but strictly spoken this is optional.

UI building implies three main classes of operations: creating new UI elements, editing existing UI elements, and reusing existing UI elements. We now detail these operations in more detail.

UI Element Description. The first operation is the definition of the elements that compose a user interface.

In many cases, developers use a graphical tool to place UI elements on a canvas. Using the tool they can edit the UI element properties such as their position, size, color, fonts used, order of tab focus, and so on. In addition, in some cases they can specify how the UI reacts to events, *e.g.,* mouse enter events. The output of this phase can be either automatically generated code or a higher level description of the user interface.

The output which is produced has an influence on what is subsequently possible. Indeed, there is a dual role here: on the one hand the UI is the elements with which the enduser will interact, but on the other hand it is also the representation of such information such that tools can manipulate it.

UI Specification Manipulation. The way UI element descriptions are stored is an important point to consider. The creation of a user interface is only one part of the development cycle. Another part is later edition of this description, *e.g.,* when evolving the application, and hence the question to consider is is how a tool can manipulate such information.

When the UI element description phase output is plain code, it is often more difficult to edit this description because it needs to be parsed to extract information. This happens for example in WindowBuilder, the google toolkit

for Java [**?**]. When a more declarative syntax is used (for example XML as in the case of NetBeans) [**?**], its manipulation is more straightforward but still requires a specific interpretation phase to create the rendered interface.

Reuse and Composition. A last, key, aspect of user interface building is how to be able to reuse an existing user interface to build a new one. To the best of our knowledge, only few approaches provide adequate support for this.

Composing an user interface is not just about spatially positioning graphical elements. It's also about specifying the relationships between those elements. In addition, some widgets are often used together. Hence being able to define a widget as a group of existing widgets is the foundation to provide composable and reusable widgets. When having such reusable widgets it is also important to produce widgets that offer specific behavior while at the same avoiding to duplicate code and maintenance.

There are several aspects to consider depending on the level of possible reuse.

- *Widget description reuse.* By widget description reuse, we mean that to build a new widget it is often necessary to be able to reuse the structure and behavior of a given widget. For example, a widget with an input field filtering a list of items can be reused in different other widgets.
- *Adapting the description.* Often it is important to be able to not only adapt the structure, *i.e.,* location, dimension, color . . . of a widget, but also its behavior. For example, the behavior of a list when the selected item is removed can be either to jump to the top of the list or to go to the next item.
- *Changing the underlying model.* Some of the properties to be adapted are purely at the level of the widget. However, there are cases where reuse implies needing to associate different UI models or to use different parts of the model. Offering reusable widgets this implies that the programmer needs to have access to the model controlling the user interface elements.

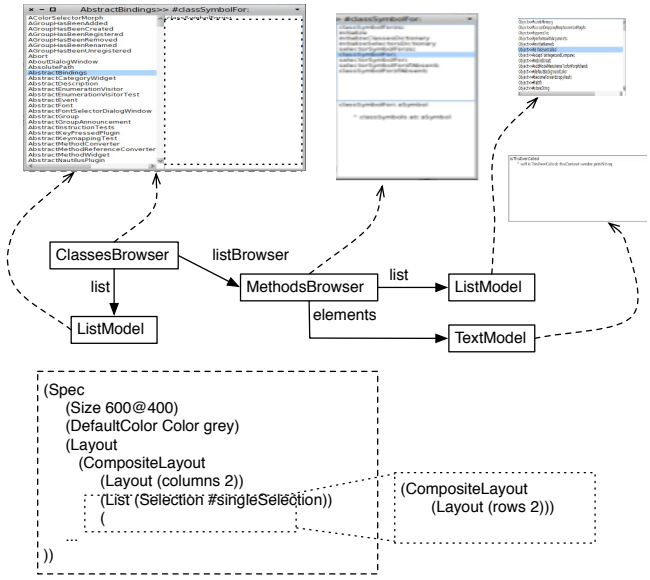

Figure 1: Spec principles: a UI is based on a class that defines the application logic and glues visual components or other application classes together using value holders.

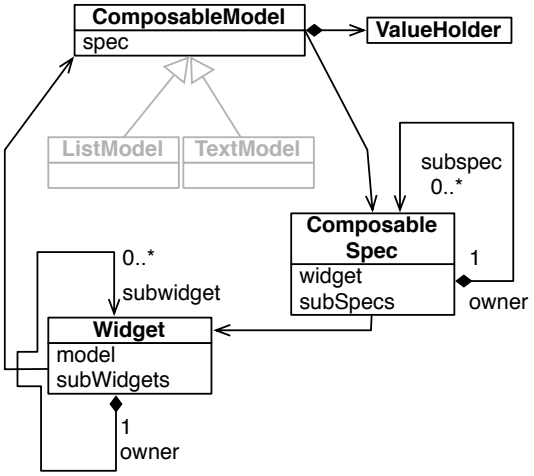

Figure 2: Relationships between composable models, widgets and specifications

3. Spec

We built Spec to support the definition and reuse of user interfaces in Pharo. 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 composition points, expressed as value holders[1], and whose visual characteristics are defined using a declarative specification that is reused when being composed. In addition to the above, each widget provides a set of interactive events, *e.g.,* a list element is selected, that other widgets can register to. By doing this, an event-based user interface can be defined where multiple widgets are bound together to define an overall UI behavior.

With Spec, widget properties are defined declaratively (as described in the next section) and attached to specific classes known as composable classes (ListModel, TextModel, MethodsBrowser in Figure 1). 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 reuse of their connection to the underlying model. In addition during the composition, layout information of the reused parts can be redefined in the context of the composite. This support for reuse is the most important contribution of Spec, as its use removed a high amount of code duplication in Pharo 2.0.

Figure 2 shows that a ComposableModel notably holds a group of ValueHolders. A value holder is a wrapper around a value that notifies its dependents of changes to the value [**?**]. In Spec, value holders are used to support the manipulation and sharing of domain models and to act as ports for the UI. Value holders can hold two different types of data: UI related state or applicative model data. A ComposableModel also holds a ComposableSpec, which can contain a number of other ComposableSpec instances and is associated to a widget. A Widget is composed of sub widgets and has an ComposableModel as model. In addition, there is a binding used by the UIBuilder to define which widget is associated to a given specification.

Reuse in Spec occurs when a ComposableSpec contains a number of other ComposableSpec instances. In the simplest case, this boils down to one widget (the mother widget) using the other one as a sub-widget (daughter widget). In all reuse scenarios the mother widget can specify the view and behavior of all daughters. Moreover, the mother widget can modify the view and behavior of all daughter widgets' sub-widgets as well, to ensure that the whole sub-widgets tree fits the required scenario. This is illustrated in Figure 3, where the mother widget changes the layout of the daughter widgets' sub-widgets, and will be discussed in more detail in Section 4.3.

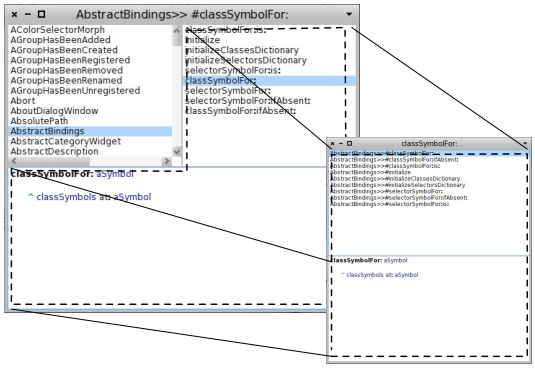

Figure 3: The classes browser reusing the methods browser and adapting its layout.

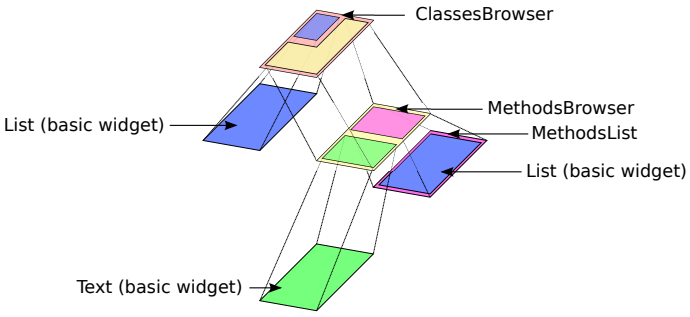

Figure 4: A ClassesBrowser is composed of a list and a MethodsBrowser. A MethodsBrowser is composed of a Text and a MethodsList, which itself is composed of a List.

The Spec runtime engine acts as a Builder design pattern [**?**]: Once the specification of the UI is given in the declarative specification and some glue methods, the runtime engine creates a fully functioning widget that is presented to the end-user. It does this by glueing UI elements, provided by the underlying windowing toolkit, to the given UI models.

[1] A value holder is an object holding a value and notifying its dependents on change.

4. Spec by Example

In this section we show an example of the typical use of Spec. We highlight how it is built on and supports reuse. Specifying UI models essentially boils down to providing four kinds of information for a UI model:

1. A UI *model* class that has storage for the embedded UI models. This class must extend the ComposableModel class, the root of the Spec UI model hierarchy. It stores its embedded UI models in a *ValueHolder* for each embedded model. Typically these are stored in instance variables, but this is not required. What is required is that for each embedded UI model an accessor is provided.

2. An initialization process for its embedded UI models. This is performed in three different methods. First an initializeWidgets method to instantiate and configure the embedded UI Models, which creates the widgets when the UI is opened. Second, an initializePresenter method to wire the different embedded UI Models together, which corresponds to specifying the presenter logic. Third, an initialize method for any UI application state.

3. A specification of the layout of its embedded UI models. This is done by implementing a method on the class side of the UI model that returns a SpecLayout object. As multiple layouts per UI model can be present, these must be tagged with a pragma <spec:>. The keyword default may be used in the pragma to specify that this is the default. If there is no <spec: #default> tag, then the method named defaultSpec must return the layout object.

4. A public API of the model such that it is cleanly reusable by other UI models. All UI model classes are embeddable in other UI model classes, i.e. they are reusable by default. Providing a public API of the relevant functionality of this model allows clean reuse of this component, avoiding the need to manipulate its internals when reusing it.

To go more in detail, we proceed with an example. We start with showing how a basic UI model can be reused to build a domain-specific UI model. This domain-specific UI model itself is then reused and composed with a generic UI model to build a complete UI. This UI is itself again a UI model, which is reused and adapted to build a last UI that is more substantial (see Figure 8). In total we build three UI models: a *Method List* that shows a collection of methods, in Section 4.1, a *Method Browser* that reuses the list and adds a pane showing the source code of the method, in Section 4.2, and lastly a *Class Browser*, reusing and adapting the *Method Browser*, in Section 4.3.

Note that we only focus here on the logic of the building process, the discussion of a beginning of a graphical tool to compose widgets based on Spec is in Section 5.3.

4.1. Methods List

As a first example of the use of Spec we create a simple domain-specific UI model called MethodsList. It is a list that displays method signatures. This done by reusing a generic list UI model, stored in a list instance variable, and adding a minimal amount of domain logic.

First the class MethodsList needs to be defined as a subclass of ComposableModel. As there is no presenter logic, only the embedded UI models have to be specified and configured in the initializeModels method, as shown below.

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

The code shows how the UI model for the list is instantiated, in a declarative fashion. The method instantiateModels: takes as argument a collection of associations where the key is the instance variable name and the value is a UI model class. The code above creates a new instance of the basic list UI model, called ListComposableModel, wraps it in a value holder, and stores it in the instance variable *list*.

By default, the method printString is sent to each item in the list to produce the string used to display the item. Often the default behavior displays too much information, or is not accurate enough. To address this, a block can be used to specify the domain-specific logic of how to generate the displayed string. This is achieved by sending the message displayBlock: to the list widget. This is shown in the second statement of the method that specifies a block such that the displayed string follows the form class name»selector.

As MethodsList reuses an existing UI model, it must specify the layout of the UI element corresponding to this model. This is shown in the following method:

```
MethodsList class>>spec
    <spec: #default>
    ^ SpecLayout composed add: #list; yourself.
```

The code above 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 UI 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 list, the selector of the getter of the embedded 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.

This completes the construction of the UI model. However if we want this UI model to be straightforwardly reusable in other UIs we should provide a public API for it. The implementation of the API is shown in Figure 5. We want to have the API of MethodsList to be polymorphic to the core protocol of the embedded list. Hence for each method of the list API we forward the message to the *list* instance variable.

```
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
```

Figure 5: The public API of MethodsList

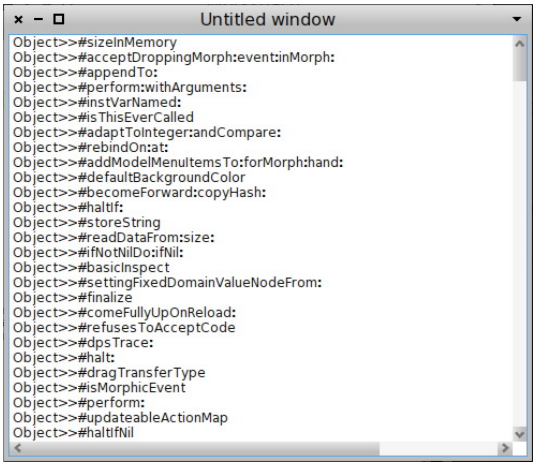

Figure 6: The methods list widget embedded in a window.

All widgets created using Spec can be embedded in a window and shown to the end-user, simply by instantiating them and sending them the openWithSpec message.

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

The code above also shows the use of the public API of MethodsList to specify a collection of items to be shown: the methods of the class Object. The result of this is shown in Figure 6.

4.2. Methods Browser: Composing a MethodsList and a TextModel

To illustrate how Spec allows for the combination and reuse of existing models and UIs, the MethodsBrowser shows a list of methods together with the source code of the selected method. To do this it embeds the methods list, kept in a methodsList instance variable, and a text field, kept in a text instance variable.

The initialization process first consists of associating these UI models to their instance variables.

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

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

Next, the overall behavior of the UI is specified by 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 links the list to the text field by stating that when an item in the list is selected the text of the text field is set to a an empty string if no item is selected, or the source code of the selected item otherwise.

After specifying the behavior of the UI, the layout of the different embedded UI models need to be specified, which is performed as follows:

```
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
```

The above method first shows that the reuse of a custom UI model, instead of a standard Spec UI model, is completely transparent with regard to the layout. Secondly, it also shows that a relative 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 layouts of UI elements can be expressed in terms of rows and columns, a simpler way to describe these layouts 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
```

Window Title. When embedding a widget in a window, in many UIs the title of the window is updated depending on the state of the widget. Spec allows for this as well, by defining the method named title on the UI model. This method must return a string, which will be the title of the window, for example as shown below.

```
MethodsBrowser>>title
    ^ methodsList selectedItem ifNil: [ 'Methods browser' ] ifNotNil: [:method | method selector ].
```

The code above returns the selector of the selected method, or the string 'Methods Browser' if no item is selected. Note that it uses the public API of MethodsList (here selectedItem). The above code however does not implement the logic for updating the title when a new method is selected, it only specifies what the current title should be. To trigger an update of the title, the message updateTitle must be sent to self.

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 is performed by amending the initializePresenter method as follows:

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

The result of opening the MethodsBrowser on the methods of Object and selecting a method is shown in Figure 7.

Recall that to have a cleanly reusable UI model, we need to define its public API. For the sake of brevity, we do not specify such an complete API here. Instead, for the sake of the example, we suppose the presence of the following methods: items:, displayBlock:, resetSelection, selectedItem. Each of these methods basically delegate to the embedded methods list.

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

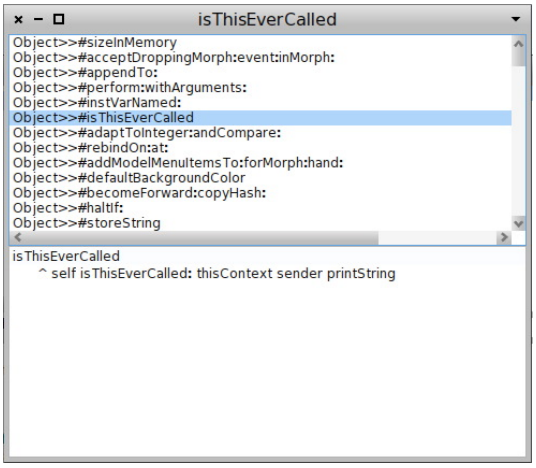

Figure 7: The methods browser

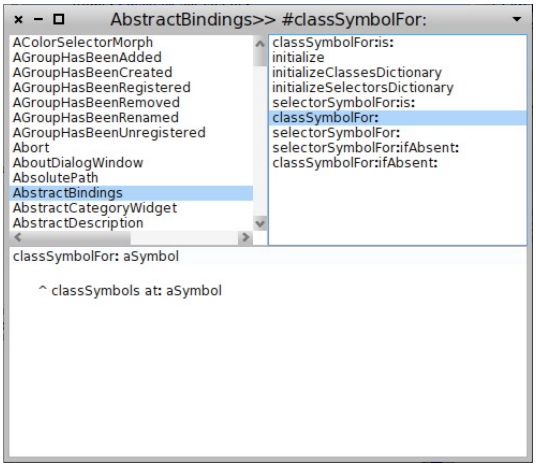

Figure 8: The classes browser, reusing the methods browser and adapting its layout to use a classic layout

4.3. Classes Browser: Composing a List and a MethodsBrowser

The last example shows how to parametrize the reuse of UI models, and how a UI communicates with the underlying model. We do this by illustrating how to build a simple class browser, shown in Fig. 8, that reuses the MethodsBrowser. Being able to deeply parametrize UI models allows for extended reuse possibilities of the models since they can be more generic.

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.

4.3.1. UI Parametrization

We first present how to parametrize the behavior of reused UI models that are embedded in a reusing 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 that UI model.

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 a default title 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.

```
ClassesBrowser>>initializeWidgets
"... omitting code for the sake of brevity ..."
    methodsBrowser displayBlock: [:method | method selector ].
```

```
ClassesBrowser>>initializePresenter
"... omitting code for the sake of brevity ..."
    methodsBrowser whenSelectedItemChanged: [ self updateTitle ].
```

Recall that the methods we are calling are part of the API for the methods browser we defined previously. 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 (which we omit, for the sake of brevity).

4.3.2. Layout Specification

The classic layout for a class browser is to have the two lists: classes and methods in a row in the top-most half of the window, and the text zone in the bottom-most half. The methods browser defined above however already defines the layout of its list and text zone, and this should therefore be overridden (as shown in Figure 3). To do this, the class browser provides the following layout method:

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

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

In the above code, when embedding the methods browser, instead of using its layout we define precisely how we want the sub-widgets to be laid out.

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

4.3.3. 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 since the classes browser reuses the methods 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 specifies 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 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. We subscribe to these announcements, for example using the code below. As a result, 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.

5. Going Beyond Static Spec

Using a dynamic system impacts directly on the way programs are created as well as the way user interfaces are designed and how objects are manipulated. Taking this into account implies providing a UI framework that fits into this dynamic world. In this section, we will first present how the Spec layouts can be dynamically changed and specified. Second we will show how to retrieve information about the customization points of widgets. And third we will introduce a tool which combines the two previous aspects to provide a dynamic tool to edit Spec widgets.

5.1. Dynamic Spec

Having a static description of a UI is sufficient for many kinds of applications, but some UIs need to be dynamically changed in accordance with the user interactions. A typical example of such a feature is a UI presenting data, and according to what the user specifies (by ticking/un-ticking options) adding or removing columns. A mechanism like this can be found in file system explorers, where the column to be displayed can be changed dynamically. Spec provides for such a feature by allowing a widget to redraw itself using another layout, and to specify which sub-widgets have to be kept and which ones need to be rebuilt.

To illustrate the dynamic showing and hiding, we show the relevant code of a contact list UI where the user can specify which columns of name, last name and age to display. For the sake of the example, suppose that the method #updateColumns of the UI Model is called whenever a checkbox is ticked or un-ticked (as specified in the initializeWidgets method of the model). In this method we first need to specify which widgets need to be rebuilt the next time the UI is built, *i.e.,* will the build create a new widget or will it reuse the existing one. Second we trigger a rebuild of the UI:

```
updateColumns
    self needRebuild: false.
    nameList needRebuild: false.
    surnameList needRebuild: false.
    ageList needRebuild: false.

    self buildWithSpecLayout: self dynamicLayout
```

Here none of the sub widgets needs to be rebuilt nor does the widget itself. The last line of the example triggers the model to rebuild itself using a layout that will be dynamically created to fit the current state. As the #dynamicLayout method below shows, the layout created displays only the lists associated to the ticked checkboxes.

```
dynamicLayout

    ^ SpecLayout composed
        newRow: [:row |
            row add: #boxesList width: 150.
            nameBox state ifTrue: [ row add: #nameList ].
            surnameBox state ifTrue: [ row add: #surnameList ].
            ageBox state ifTrue: [ row add: #ageList ]]
```

In addition to the dynamic layout, Spec also provides a mechanism to allow the widget to dynamically instantiate new sub-widgets. For example this is needed when creating a UI for a tool that performs searches. In this tool, the user can add as many filters as he wants. In this case, the programmer can't predict how many filters will be added. So it is up to the widget to handle this.

The following code shows, for this example, how to dynamically instantiate a new sub widget and how to use it. Note that the UI Model class of the search tool must inherit from DynamicComposableModel to enable this functionality. The method #instantiateModels: will create the new sub widget and store it. This sub widget can then be accessed via a getter automatically made available by the Spec infrastructure. In addition, the method #needFullRebuild, informs all the dynamically added sub widgets that they don't need to be rebuilt.

```
addANewFilterWidget
    self needFullRebuild: false.

    "note that the new sub widget name must be unique"
    self instantiateModels: {#myNewFilter MyFilterWidgetClass}.
    self myNewFilter state: true.
    self buildWithSpecLayout: self dynamicLayout
```

The two mechanisms detailed above: dynamic showing and hiding of widgets, and dynamically adding and removing widgets, can of course be linked together. This allows widgets to be even more modular since the layout itself can be specified depending on the model data.

5.2. Model and widget Metadata

To support the extension of the set of composable models as well as their graphical composition, well-structured metadata is present in each of these classes. For example, methods playing specific roles during the composition are tagged with meta-descriptions. All basic widget methods and all ComposableModel subclasses use the same structure and mechanism to provide API meta information. This information is programmatically retrievable, which allows Spec tools to have a more precise representation of the widgets as well as supporting the addition of new models and widgets. Lastly, this provides documentation to the user about how to use a widget by tagging all the interface methods, and providing information about how to use them.

The metadata is specified using a pragma that has the following structure: first the api: keyword is used to flag the method as providing metadata. The argument for this keyword is a symbol that specifies the type of the value for the customization point. This is followed by additional information about the value, *e.g.,* the range it belongs to. Next-to-last, the getter: keyword takes as argument the symbol for the selector used to retrieve the current value. And last, the registration: keyword takes as argument the selector to use to register to changes to the customization point.

For example, the figure 9 shows the metadata attached to the method #setSelectedIndex. This data contains multiple pieces of information. First the #integer keyword specifies that the argument of the method is an object that behaves as an integer. Second, the keywords #min and #max provide the range of the argument. The #max argument is the selector to use to retrieve this information. Third it shows that the selector to retrieve the current value of the selection index is #selectedIndex. And fourth it details that the selector to register an object to the changes of the index is #whenSelectionIndexChanged:.

```
setSelectedIndex: anIndex
    <api: #integer min: 0 max: #listSize getter: #selectedIndex registration: #whenSelectionIndexChanged:>
```

Figure 9: Metadata of the method ListModel»#setSelectedIndex:

5.3. Live Editor

Spec provides for a live editor of widget properties that relies on the dynamic layout and the meta-information of the widgets' protocol. This editor allows users to access and edit Spec widget properties live. Each widget customization point is represented in the editor, and each modification performed on this customization point impacts the inspected widget directly. This allows a user to create a widget and then inspect it. The user has access to all its properties, and can dynamically set them to the required values. This tool can be used to simply edit the current state of a widget for experimenting, but it also provides a fast and simple way to prototype UIs. Note that a mechanism to reflect changes made in this tool back to the code is currently absent.

Figure 10 shows the an editor on a multi column list. The #setSelectedIndex from the previous example can be found near the bottom of the window. The fact that the argument to this method is an integer has been used to produce a slider widget to change this integer easily. In addition, the #min and #max specifications have been used to set the range of this slider. Consequently, the slider can not set a value smaller than 0 or larger than the value returned by the method #selectedIndex. And finally, this slider registered itself to the index modification using the argument of the #registration meta-data element. To sum up, all the information provided by the meta-data has been used to provide an accurate widget reflecting the current state of the customization point and offering a simple way to impact the current state of this customization point.

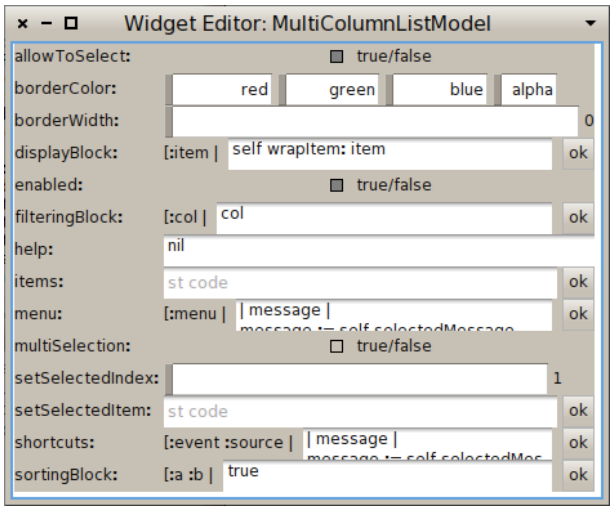

Figure 10: Living editor on a multi columns list

6. The Implementation of Spec

The implementation of Spec is based on two pillars: the presence of a Spec object and the use of value holders. The Spec is composed of a SpecLayout which 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.

6.1. 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 4.1. 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.

6.2. 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 to specific changes to the model. As such, we provide an event based structure that allows one to react to only to value changes 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 [**?**] and MVP [**?**]. 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 with each observable's value holder in which it is interested, 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 ensures that there are no event loops due to circular event sends.

Note that since every object can register for a value holder's changes, this means that a model can register itself with any of its sub-widgets' value holders, or any sub-widget's sub-widget's 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 fit a new scenario.

7. 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

7.1. 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 [**?**]. 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.

7.2. 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.

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: readSelection codePaneMenu:shifted:);
            send: #classOrMetaClass: withArguments: #(model behavior);
            send: #enabled: withArguments: #(model enabled);
            send: #eventHandler: withArguments: #(EventHandler on:send:to: keyStroke keyStroke:fromMorph: model);
            send: #vSpaceFill;
            send: #hSpaceFill;
            yourself
```

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

7.3. Dynamic Spec

Dynamic Spec refers to the capability of a model to dynamically update its layout as well as its set of sub-widgets, as seen in section 5.1. The first feature can be applied to any Spec widget and needs to use two specific methods: #needRebuild: and #buildWithSpecLayout:. These methods are detailed in the first half of the table 4. The second feature, provided by the class DynamicComposableModel, is a mechanism to dynamically add new sub-widgets or to remove them. The mechanism is based on the two methods #instantiateModels: and #deInstantiateWidget: as shown in the second half of the table 4.

Selector	Result
ComposableModel»needRebuild:	specifies whether or not the current model's widget has to be reused during the next build.
ComposableModel»buildWithSpecLayout:	build the receiver using the SpecLayout instance provided as argument.
DynamicComposableModel»instantiateModels:	instantiate a list of sub widgets using the model class provided and store them into a dynamic data structure of this model.
DynamicComposableModel»deInstantiateWidget:	remove the sub widget provided as argument.

Table 4: Dynamic Spec public API

8. Spec in Pharo

Spec has been introduced in Pharo 2.0 with the goal of being used for rewriting all the tools. For now, six different tools have been re-implemented:

Tool	Pharo 1.4 (without Spec)	Pharo 2.0 (with Spec)	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 5: LOC Comparison of tools with and without Spec

1. MessageBrowser: a tool made for browsing messages (similar to the MethodsBrowser made in section 4.2);
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;
5. ChangeSorter: a tool made for managing changes to 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 ChangeSorters linked together and specialized to add functionality involving the interactions between the two change sorters.

Table 5 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 adds new features like a topological sort, an update mechanism, and support for methods in addition to 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 lines 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 as for the 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 reduce 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.

9. Related Work

Spec is inspired by the VisualWorks [**?**][**?**] 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 [**?**] 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 [**?**] or WindowBuilder [**?**] 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 to the UI code, Spec also provides a framework for the model behavior while NetBeans and WindowBuilder only provide UI element 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 [**?**] is an XML based language used for describing and reusing 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 [**?**] is a DSL based on the strategy transformation framework that proposes a more declarative syntax for expressing widget description in Swing. SWUL behaves like XUL in the sense of not being able to locally redefine the behavior of a sub-widget.

Microsoft ASP.NET is a web application framework developed by Microsoft [**?**]. ASP.NET provides an implementation of the MVP pattern. The main difference between the Dolphin implementation (discussed in Section 2.1) and ASP.NET is the description of the views. A view is declared in a web form, *i.e.,* as a collection of (X)HTML declarations. These declarations describe the composition of the views and how the data are presented. As with the Dolphin implementation, the binding between a view declaration and a presenter relies on name matching.

10. 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 being able to build new widgets.

To address this, we have built Spec and we report on it in this paper. After a discussion of challenges of UI Builders we provided a general introduction to Spec. We have then shown how Spec can be used, by providing three example UIs that highlight the reuse and parametrization features of Spec. This was followed by an overview of the more dynamic features of Spec. We next talked about the implementation of Spec and provided a more formal specification of the APIs used in the example. Lastly, we showed how Spec enabled a 57% code reduction in the re-implementation 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/).

References