Creating new basic widgets

Spec provides for a large amount and wide variety of basic widgets. In the rare case that a basic widget is missing, the *Spec* framework will need to be extended to add this new widget. In this section we will explain how to create such a new basic widget.

We will first explain the applicable part of how the widget build process is performed. This will reveal the different actors in the process and provide a clearer understanding of their responsibilities. We then present the three steps of widget creation: writing a new model, writing an adapter, and updating or creating an individual UI framework binding.

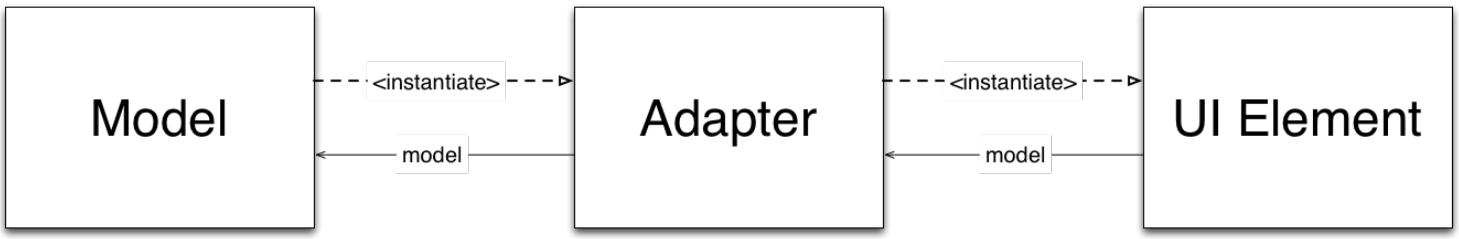
One step in the building process of a widget

The UI building process does not make a distinction between basic and composed widgets. Hence, at a specific point in the building process of a basic widget the default spec method of the widget is called, just as if it would be a composed widget. However in this case, instead of providing a layout for multiple widgets that comprise the UI, this method builds an adapter to the underlying UI framework. Depending of the underlying UI framework that is currently used, this method can provide different kind of adapters, for example an adapter for Morphic, or an adapter for Seaside, etc.

The adapter, when instantiated by the UI model, will in turn instantiate a widget that is specific to the UI framework being used.

For example, when using a List in the Morphic UI, the adaptor will be a MorphicListAdapter and it will contain a PluggableListMorph. This is this framework-specific widget that will be added to the widget container.

Figure 1.1 shows the relationship between those objects.



The Model

The new model needs to be a subclass of **AbstractWidgetModel** and its name should be composed of the new basic widget concept, e.g. list or button, and of the word *Model*. The responsibility of the model is to store all the state of the widget. Examples of widget-specific state are:

- the index of a list
- the label of a button
- the action block for when a text is validated in a text field

The state is wrapped in reactive variables and kept in instance variables. For example, the code following shows how to wrap the state `0` in a reactive variable and keep it as an instance variable. reactive variables are needed because they are later used to propagate state changes and thus create the interaction flow of the user interface, as discussed in the page Sub widgets interaction (/docs/interactions/).

```
index := 0 asValueHolder.
```

For each instance variable that holds state three methods should be defined: the getter, the setter, and the registration method. The first two should classified in the protocol named **protocol** while the registration method should be in **protocol-events**.

For example, the methods to define for the previous example can be implemented as following:

```
"in protocol: protocol"
index
    ^index value

"in protocol: protocol"
index: anInteger
    index value: anInteger

"in protocol: protocol-events"
whenIndexChanged: aBlock
    index whenChangedDo: aBlock
```

The last step to define a new model is to implement a method `adapterName` at the class side. The method should be in the protocol named *spec* and should return a symbol. The symbol should be composed of the basic concept of the widget, e.g. list or button, and the word **Adapter** like **ListAdapter**.

The communication from the UI model to the adapter is performed using the dependents mechanism. This mechanism is used to to handle the fact that a same model can have multiple UI elements concurrently displayed.

The message `changed: with:` is used to send the message selector with the arguments aCollection to the adapters. Each adapter can then convert this Spec message into a framework specific message. For example, the method `#filterWith:` sent by **TreeModel** via `changed: with:` is implemented as:

```
filterWith: aFilter

   self widgetDo: [ :w || nodes |
           nodes := w model rootNodes.
           nodes do: [ :r | r nodeModel updateAccordingTo: aFilter ].

           self removeRootsSuchAs: [ :n | (aFilter keepTreeNode: n) not and: [ n
sEmpty ] ].

           self changed: #rootNodes ].
```

The Adapter

An adapter must be a subclass of **AbstractAdapter**. The adapter name should be composed of the UI framework name, e.g. Morphic, and the name of the adapter it is implementing, e.g. ListAdapter. The adapter is an object used to connect a UI framework specific element and the framework independent model.

The only mandatory method for an adapter is `defaultSpec` on the class side. This method has the responsibility to instantiate the corresponding UI element.

The next example shows how **MorphicButtonAdapter** instantiates its UI element.

```
defaultSpec
    <spec>

    ^ #(PluggableButtonMorph
            #color:                             #(model color)
            #on:getState:action:label:menu:     #model #state #action #label nil
            #getEnabledSelector:                #enabled
            #getMenuSelector:                   #menu:
            #hResizing:                         #spaceFill
            #vResizing:                         #spaceFill
            #borderWidth:                       #(model borderWidth)
            #borderColor:                       #(model borderColor)
            #askBeforeChanging:                 #(model askBeforeChanging)
            #setBalloonText:                    #(model help)
            #dragEnabled:                       #(model dragEnabled)
            #dropEnabled:                       #(model dropEnabled)
            #eventHandler:                      #(EventHandler on:send:to: keySt
rph: model))
```

Since the adapter is bridging the gap between the element of the UI framework and the model, the adapter also needs to forward the queries from the UI element to the model. Seen from the other way around: since the model is holding the state, the adapter is used to update the UI element state of the model.

The methods involved in the communication from the model to the adapter as well as the methods involved in the communication from the adapter to the UI model should be in the protocol spec protocol. On the other hand the methods involved in the communication from the adapter to the UI element and vice versa should be categorized in the protocol widget API.

To communicate with the UI element, the adapter methods use the method `widgetDo:`. This method executes the block provided as argument, which will only happen after the UI element has already been created.

The follwoing example shows how **MorphicLabelAdapter** propagates the modification of the emphasis from the adapter to the UI element.

```
emphasis: aTextEmphasis

    self widgetDo: [ :w | w emphasis: aTextEmphasis ]
```

The UI Framework binding 🔗

The bindings is an object that is used to resolve the name of the adapter at run time. This allows for the same model to be used with several UI frameworks.

Adding the new adapter to the default binding is quite simple. It requires to update two methods: `initializeBindings` in **SpecAdapterBindings** and `initializeBindings` in the framework specific adapter class, e.g. **MorphicAdapterBindings** for Morphic.

The method `SpecAdapterBindings>>#initializeBinding` is present only to expose the whole set of adapters required. It fills a dictionary as shown in the code:

```
initializeBindings
	"This implementation is stupid, but it exposes all the containers which ne
 bound"

	bindings
		at: #ButtonAdapter			put: #ButtonAdapter;
		at: #CheckBoxAdapter		put: #CheckBoxAdapter;
		at: #ContainerAdapter		put: #ContainerAdapter;
		at: #DiffAdapter			put: #MorphicDiffAdapter;
		at: #ImageAdapter			put: #ImageAdapter;
		at: #LabelAdapter			put: #LabelAdapter;
		at: #ListAdapter			put: #ListAdapter;
		at: #IconListAdapter		put: #IconListAdapter;
		at: #DropListAdapter		put: #DropListAdapter;
		at: #MultiColumnListAdapter	put: #MultiColumnListAdapter;
		at: #MenuAdapter			put: #MenuAdapter;
		at: #MenuGroupAdapter		put: #MenuGroupAdapter;
		at: #MenuItemAdapter		put: #MenuItemAdapter;
		at: #NewListAdapter			put: #NewListAdapter;
		at: #RadioButtonAdapter		put: #RadioButtonAdapter;
		at: #SliderAdapter			put: #SliderAdapter;
		at: #TabManagerAdapter		put: #TabManagerAdapter;
		at: #TabAdapter				put: #TabAdapter;
		at: #TextAdapter			put: #TextAdapter;
		at: #TextInputFieldAdapter	put: #TextInputFieldAdapter;
		at: #TreeAdapter			put: #TreeAdapter;
		at: #TreeColumnAdapter		put: #TreeColumnAdapter;
		at: #TreeNodeAdapter		put: #TreeNodeAdapter;
		at: #WindowAdapter			put: #WindowAdapter;
		at: #DialogWindowAdapter	put: #DialogWindowAdapter;
		yourself
```

Each framework specific adapters set should define its own binding. To implement a new binding, a subclass of **SpecAdapterBindings** must be defined that overrides the method `initializeBindings`. This method must binds Spec adapter names with framework specific adapter class names. The following example shows how the morphic binding

implements the method `initializeBindings`.

```
initializeBindings

	bindings
		at: #ButtonAdapter			put: #MorphicButtonAdapter;
		at: #CheckBoxAdapter			put: #MorphicCheckBoxAdapter;
		at: #ContainerAdapter			put: #MorphicContainerAdapter;
		at: #DiffAdapter			put: #MorphicDiffAdapter;
		at: #DropListAdapter			put: #MorphicDropListAdapter;
		at: #LabelAdapter			put: #MorphicLabelAdapter;
		at: #ListAdapter			put: #MorphicListAdapter;
		at: #IconListAdapter			put: #MorphicIconListAdapter;
		at: #ImageAdapter			put: #MorphicImageAdapter;
		at: #MultiColumnListAdapter		put: #MorphicMultiColumnListAdapter;
		at: #MenuAdapter			put: #MorphicMenuAdapter;
		at: #MenuGroupAdapter			put: #MorphicMenuGroupAdapter;
		at: #MenuItemAdapter			put: #MorphicMenuItemAdapter;
		at: #NewListAdapter			put: #MorphicNewListAdapter;
		at: #RadioButtonAdapter		put: #MorphicRadioButtonAdapter;
		at: #SliderAdapter			put: #MorphicSliderAdapter;
		at: #TabManagerAdapter		put: #MorphicTabManagerAdapter;
		at: #TabAdapter			put: #MorphicTabAdapter;
		at: #TextAdapter			put: #MorphicTextAdapter;
		at: #TextInputFieldAdapter		put: #MorphicTextInputFieldAdapter;
		at: #TreeAdapter			put: #MorphicTreeAdapter;
		at: #TreeColumnAdapter		put: #MorphicTreeColumnAdapter;
		at: #TreeNodeAdapter			put: #MorphicTreeNodeAdapter;
		at: #WindowAdapter			put: #MorphicWindowAdapter;
		at: #DialogWindowAdapter		put: #MorphicDialogWindowAdapter;
		yourself
```

Once this is done, the bindings should be re-initialized by running the following snippet of code: `SpecInterpreter hardResetBindings`.

For creating a specific binding, the class **SpecAdapterBindings** needs to be overriden as well as its method `initializeBindings`.

Then during the process of computing a Spec model and its layout into a framework specific UI element, the binding can be changed to change the output framework. The binding is managed by the **SpecInterpreter**. The next example shows how to do so.

```
SpecInterpreter bindings: MyOwnBindingClass new.
```

The **SpecInterpreter** bindings are reset after each execution.

Back (/docs/api/) Next (/docs/interpreter/)

Getting Started

Welcome (/docs/home/)
Quick start (/docs/quickstart/)
Installation (/docs/installation/)
Example (/docs/example/)

Your UI

Where to start? (/docs/starting/)
Instantiating sub widgets (/docs/initializing/)
Defining the layout (/docs/layout/)
Opening your UI (/docs/open/)
Sub widgets interaction (/docs/interactions/)
Dynamic UI (/docs/dynamic/)
Using Pharo window (/docs/use-pharo-window/)
Inserting a Morph

(/docs/insert-morph/)

Inside Spec

Where to find what I want
(/docs/api/)
Creating new basic widgets
(/docs/own-model/)
Spec Interpreter
(/docs/interpreter/)

Tutorials

Drag'n'Drop between two lists
(/docs/drag_n_drop/)

More

Repositories
(/docs/repositories/)