# Chapter 1

# A look into Spec

need an intro

Spec was developed by Benjamin Van Ryseghem, you can find more information in: https://pharo.fogbugz.com/default.asp?spec.5.5.2

## 1.1 Doing the minimun for opening a Window

We want an empty windowâĂę but our empty window, soâĂę how?

### We are going to create a class extending ComposableModel

```
ComposableModel subclass: #TestSpec
    instanceVariableNames: âĂİ
    classVariableNames: âĂİ
    poolDictionaries: âĂİ
    category: 'Flamel-UI'
```

### We have to implement 2 methods

initializeWidgets which responsibility is to setup the components we will use (right now nothing)

```
TestSpec>>initializeWidgets
    âĂIJnothing to do hereâĂİ
```

defaultSpec (class side) which returns a Layout saying which components will be render. There is a number of different layouts implemented: Spec-

ColumnLayout, SpecRowLayout, and in the class methods of SpecLayout you can find a lot more. We will choose a very easy one:

```
TestSpec class>>defaultSpec
   <spec>
   ^ SpecLayout composed
```

In fact you can name the method as you want, only need to be annotated with the <spec> pragma.

### Open the window, we evaluate in a workspace:

```
TestSpec new openWithSpec
```

and we obtain

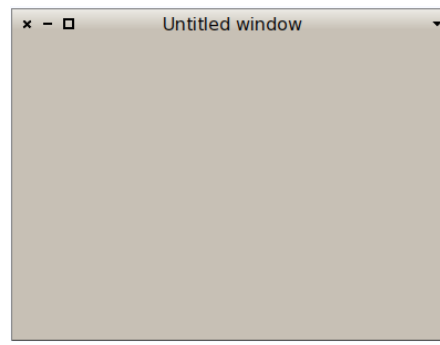


Figure 1.1: An empty Window.

## 1.2 Adding a component

1. First we have to choose the widget to add, you can see some in Spec-Widgets, we want a text area so we will choose a TextModel.

2. To render the component we need to: create it and the tell the layout that we want to render it

Let us see go over this second step

### Create the widget

Define an instance variable and instantiate it during the initializeWidget using the Spec way to do it:

```
TestSpec>>initializeWidgets
  self instantiateModels: #(text TextModel)
```

We need to define text as an instance variable because instantiateModels assume that and try the elements of the array as an association: variable - kind where variable is a property in the object and makes it point to a new instance of the kind, in order to avoid the troubles we will define the variable.

Be careful with this if you define the variable while you are debugging you can get some nasty errors, so the advice is to define the variable first and then open the window.

At this point if we get anxious to see our cool text component and we open the window we will see that is still empty, this is because we never added it to the layout! So it's not rendered.

### Add the component in the layout,

For this we will modify the defaultSpec class method and add a getter for the text property:

```
  ^SpecColumnLayout new
  add: #text;
  yourself
```

We need the getter because the layout tries to access the variable sending the symbol as a message.

### Open it

Open it and you should get a window similar to the one shown in Fig 1.2.

## 1.3 Communicating Components

Now we can open a window and have our component displayed inside. Now we want to have several components that depend on each other.

We will build a window with a text editor, when the content in the text editor changes we will see the message "The text changed" we can accept

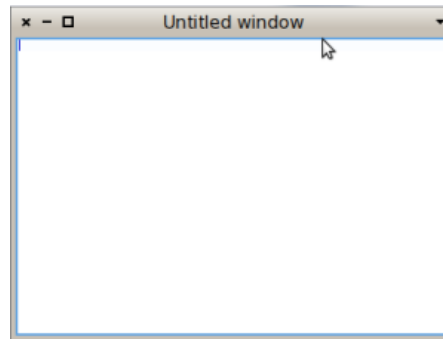


Figure 1.2: With component.

that pressing a button that says "Ok" and then the message will change to "You are up to date" (until someone change the content in the text âĂę). For doing this we will use:

- a TextModel called: example
- a LabelModel called: changes, starts with the text: "You are up to date"
- a ButtonModel called: acceptChanges, starts disabled and with the label: "Ok"

```
TestSpec>>initializeWidgets
   self instantiateModels: #(
      example TextModel
      changes LabelModel
      acceptChanges ButtonModel ).
   changes text: 'You are up to date'.
   acceptChanges label: 'Ok';  disable.
```

This is like we had before, only that we use other components that needs a little more of configuration.

The real challenge here is to link some effects: when the contents of example changes we want to change the message in changes and enable acceptChanges.

For defining this kind of interactions we should define the method initializePresenters defining the interactions, lucky our models understand several useful message to define an action to perform for an event, an example for a TextModel whenTextIsAccepted: , whenTextChanged:, ...

Let's define the actions:

```
TestSpec>>initializePresenter
   example whenTextIsAccepted: [ changes text: 'The text Changed'. acceptChanges
      enable ].
   acceptChanges action: [ changes text: 'You are up to date'. acceptChanges disable ].
```

In this part we are done, now remember that in order to render the components we need to add it to the layout:

```
   <spec>
      ^ SpecColumnLayout new
         add: #example;
         add: #changes;
         add: #acceptChanges;
         yourself.
```

And we see the situation in Fig 1.3.

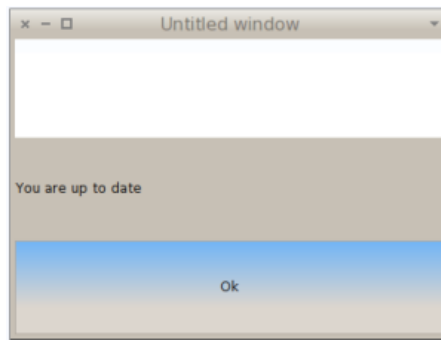


Figure 1.3: With components.

Quite easy and good, but lets improve the layout and play a little:

```
   <spec>
   ^SpecLayout composed
      newColumn: [ :mainColumn| mainColumn
         add: #example;
         newRow: [ :feedbackRow | feedbackRow
         add: #changes;
         add: #acceptChanges]];
         yourself.
```

Better but still a little detail:

```
TestSpec class>>defaultSpec
   <spec>
   ^SpecLayout composed
```

```
    newColumn: [ :mainColumn| mainColumn
      add: #example;
      newRow: [ :feedbackRow | feedbackRow
      add: #changes;
      add: #acceptChanges
      ] height: 26.];
      yourself.
```

A lot better! And It was quite easy, we can see that is well separated the layout from the components, and that we have a nice proposal for organizing our UI-Code:

1. We define the layout in defaultSpec.
2. We instantiate the components we are going to use in initializeWidget.
3. We define the interactions in: initializePresenter.

This is very nice, but now I have a UI that separates the logic but still has a lot of responsibilities, is the UI who knows if I've accepted a change or if I'm up to date, also knows exactly what to tell me... I don't think that this is the UI responsibility I want to model a class and that the UI shows it and not all in one like we have now. And how do we do it?

## 1.4 Using a Model

We will have a model behind our window in 3 easy steps:

### Create the model when initializing (you can create it or receive it, as you want, I will show the simplest way)

```
TestSpec>>initialize
  model := AcceptableText new. âĂIJreplace for your domain classâĂİ
  super initialize.
```

### define the widgets, is almost the same that before

```
TestSpec>>initializeWidgets
  self instantiateModels: #(
    example TextModel
    changes LabelModel
    acceptChanges ButtonModel ).
    acceptChanges label: 'Ok'; disable.
```

### Define the interaction

```
TestSpec>>initializePresenter
   example whenTextChanged:
      [:changed |
         model text: changed.
         self updateContents].
   acceptChanges action:
      [ model acceptChanges.
      self updateContents ]
   updateContents
   changes text: model status.
   acceptChanges enabled: model hasUnreadChanges.
```

Look that when we are defining the actions we do 2 things:

1. modify the model
2. update the content of the widgets to show the new state of the model

We can see this actions that are using blocks as controllers (in a MVC pattern) since they modify the model from an input in the view and then update the view to show the new state in the model, in other frameworks we can use Bindings that do exactly the same.

In fact Ben pointed that Spec is far more closer from a MVP pattern than an MVC In order to write a good UI we have to be very careful in how much code goes there, we probably will like to have small controllers delegating in the model instead of big controllers that does all the work for the model if we are not careful there we may end with anemic objects and spaghetti uis.

But to much talk and see how it ends:

Basically the same that before but now we have the linked ui using our domain object! Quite nice, and really really really simple to use.

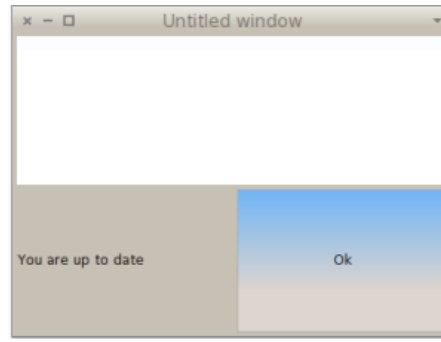


Figure 1.4: An empty Window.

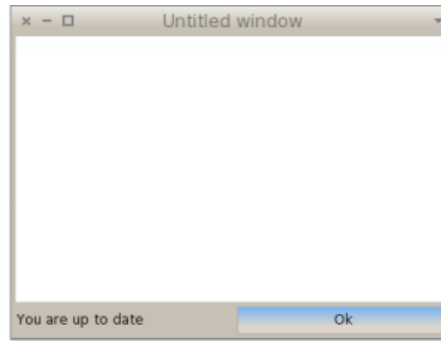


Figure 1.5: An empty Window.

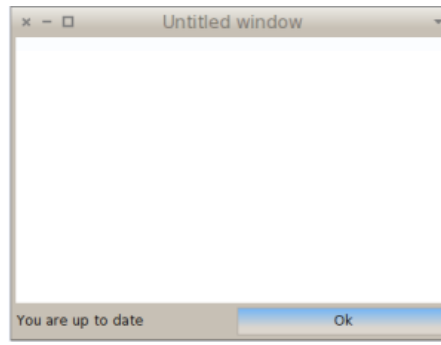


Figure 1.6: An empty Window.