Contents

CHAPTER 1

Todo Application: a 15min tutorial

status: Ready for review

This is a small tutorial to give a first look at the world of Spec2 and developing application with it. It should not be taken as a comprehensive guide since a lot of details and featurs are left out explicitly to avoid difficult issues.

We will build a small Todo application that will connect a couple of components and it will show some interesting characteristics. It will look like Figure 1-1.

1.1 A TodoTask class

Since this application will show and support the edition of a simple Todo list, we need to create a class that models the task. We are going to have the simplest approach possible for this example, but is clear it can be a lot more complex. We do not show accessor definitions.

```
Object << #TodoTask
    slots: { #done . #title };
    package: 'CodeOfSpec20BookTodo'

TodoTask >> initialize
    super initialize.
    self title: 'Please give me a title'.
    self done: false

TodoTask >> isDone
```

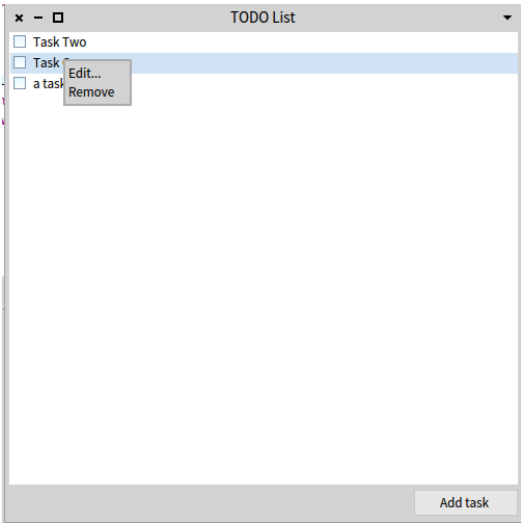

Figure 1-1 First full version of the Task List Manager.

```
    ^ self done
```

We use the class side to act as a little database.

```
Object class << TodoTask class
    slots: { #tasks }
```

We use lazy initialization to initialize the tasks collection.

```
TodoTask class >> tasks
    ^ tasks ifNil: [ tasks := OrderedCollection new ]
```

We add two methods to manage the addition and removal

```
TodoTask class >> addTask: aTask
    (self tasks includes: aTask) ifFalse: [ self tasks add: aTask ]

TodoTask class >> deleteTask: aTask
    self tasks remove: aTask ifAbsent: [ nil ]
```

We made sure that adding a task does not add twice the same task because we can use it for `save` and `accept`.

At the level of instance, we add a `save` method to register the instances in the tasks stored in the class and a `delete` method.

```
TodoTask >> save
    self class addTask: self
```

```
TodoTask >> delete
    self class deleteTask: self
```

And let's create a couple of testing tasks:

```
TodoTask new title: 'Task One'; save.
TodoTask new title: 'Task Two'; save.
```

1.2 Creating your application

Every application needs an entry point, a place where to configure the basics and start the GUI. Compiled programs have like C have `main()`, Cocoa have the class `NSApplication` and you add a *delegate* to add your configuration and Gtk3 has `GtkApplication`. Pharo is a living environment where many things can be executed at same time, and because of that Spec2 also needs its own entry point: Your application needs to be independent of the rest of system! To do that, Spec2 needs you to extend the class `SpApplication`.

```
SpApplication << #TodoApplication
    package: 'CodeOfSpec20BookTodo'
```

Note that an application is not a visual element, it manages the application and information that may be displayed visually such as icons but also other concerns such as the backend.

You will also need your main window, a Todo list. In Spec2, all components that you create inherit (directly or indirectly) from a single root base class: `SpPresenter`. A *presenter* is how you expose (present) your data to the user.

We create a presenter, named `TodoListPresenter` to represent the logic of managing a list of Todo items.

```
SpPresenter << #TodoListPresenter
    package: 'CodeOfSpec20BookTodo'
```

This component will contain a list of your Todo tasks and the logic to add, remove, or edit them. Let's define your first presenter contents.

1.3 Showing tasks

A presenter needs to define a *layout* (how the component and its subcomponents will be displayed) and which *widgets* it will show. While this is not the best way to organise your presenter, for simplicity we will add all needed behavior in just a single method that you need to implement: `initializePresenters`.

```
TodoListPresenter >> initializePresenters
    todoListPresenter := self newTable
        addColumn: ((SpCheckBoxTableColumn evaluated: [:task | task
    isDone]) width: 20);
        addColumn: (SpStringTableColumn title: 'Title' evaluated:
    [:task | task title]);
```

```
        yourself.
    self layout: (SpBoxLayout newTopToBottom
        add: todoListPresenter;
        yourself)
```

In general it is better to define a separate method `defaultLayout` whose job is to only describe the layout of the presenter.

Even if we want to manage a list of tasks, we use a table because we want to display multiple information side by side. In this case, you are adding to your presenter a table widget, which is a very complex component by itself. Let us explain what each part of it means:

- `newTable` is the factory method that creates the table component that you are going to use to display your Todo list.
- `addColumn:` is the way you add different table columns (you can have several, if we wanted to have just a single string we would have use a list).
- `SpCheckBoxTableColumn evaluated: [:aTask | aTask isDone]` creates a table column that displays the status of your Todo task (done or not done).
- `width: 20` is to avoid the column to take all available space (otherwise, the table component will distribute the available space proportionally by column).
- `SpStringTableColumn title: 'Title' evaluated: [:aTask | aTask title])` is the same as `SpCheckBoxTableColumn` but it creates a column that has a title and it will to show the title of the task as a string.

And about the layout definition:

- `SpBoxLayout newTopToBottom` creates a box layout that distributes elements in vertical arrangement.
- `add: todoListPresenter` adds the list as a subcomponent of our presenter. Here we only have one but you can have as many subcomponents you want. Note that by default, the box layout distributes all elements in proportional way (since this is the only element for the moment, it will take 100% of the avalaible space.

And now, we need to give our Todo list the tasks to display:

```
TodoListPresenter >> updatePresenter
    todoListPresenter items: TodoTask tasks
```

1.4 How does this look?

Now defining the method `start`, we tell the application that when ask to run it should open the todo list presenter.

```
TodoApplication >> start
    TodoListPresenter open
```

Now we can open our task list manager as follows and we should get a situation similar to the one displayed in Figure 1-2.

```
TodoApplication new run.
```

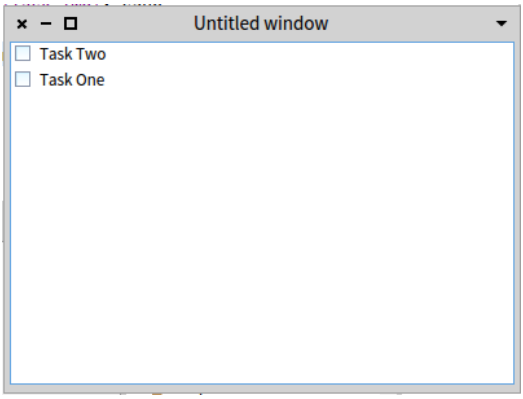

Figure 1-2 A draft version of our task manager.

Not bad as a start, isn't? But you will see the window has "Untitled window" as title, and maybe its size is not right.

To fix this we implement another method `initializeWindow:`, which sets the values we want.

```
initializeWindow: aWindowPresenter

    aWindowPresenter
        title: 'Todo List';
        initialExtent: 500@500
```

Here we took `aWindowPresenter` (the presenter that contains definition of a window), and we add:

- `title:` which is self explanatory, it sets the title of the window
- `initialExtent:` it declares the initial size of the window.

You may ask why this is done like that and not directly modifying the presenter? And the answer is quite simple: Presenters are designed to be reusable.

This means that a presenter can be used as a window in some cases (as ours), but it can be used as a part of another presenter in other. Then we need to split its functionality between *presenter* and *presenter window*.

Figure 1-3 shows how the task manager looks like now.

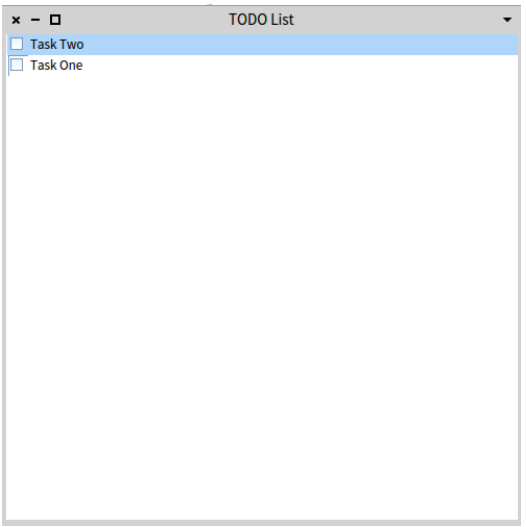

Figure 1-3 Task Manager is a better window title.

1.5 Making checkbox columns actionable

If you play a little bit with your newly created presenter, you will notice that if you enable the task checkbox and you re-execute the application (reopen the window), nothing changes in the task, it is still marked as not done. This is because no action is yet associated to the checkbox column! To fix this, let's modify `initializePresenters`.

```
TodoListPresenter >> initializePresenters

    todoListPresenter := self newTable
        addColumn: ((SpCheckBoxTableColumn evaluated: [:task | task
    isDone] )
            width: 20;
            onActivation: [ :task | task done: true ];
            onDeactivation: [ :task | task done: false ];
            yourself);
        addColumn: (SpStringTableColumn title: 'Title' evaluated:
    #title);
        yourself.
```

```
    self layout: (SpBoxLayout newTopToBottom
        add: todoListPresenter;
        yourself)
```

We added

- `onActivation:/onDeactivation:` are messages which add behavior to make sure our task gets updated.

Now you will see everything changes as expected.

1.6 Adding new tasks to your list

Now, how do we add new elements? We will need different things:

1. Modify your presenter to add a button to allow you the creation of tasks.
2. A dialog to ask for your new task.

Let's create that dialog first. Again we define a new subclass of `SpPresenter` named `TodoTaskPresenter`.

```
SpPresenter << #TodoTaskPresenter
    slots: { #task . #titlePresenter };
    package: 'CodeOfSpec20BookTodo'
```

```
TodoTaskPresenter >> task: aTask
    task := aTask.
    self updatePresenter
```

This is very simple, the only thing that is different is that setting a task will update the presenter (because it needs to show the contents of the title). And now we define initialization and update of our presenter.

```
TodoTaskPresenter >> initializePresenters

    titlePresenter := self newTextInput.

    self layout: (SpBoxLayout newTopToBottom
                add: titlePresenter expand: false;
                yourself).
```

This is almost equal to our list presenter, but there are a couple of new elements.

- `newTextInput` creates a text input presenter to add to our layout.
- `add: titlePresenter expand: false`: Along with the addition of the presenter, we also tell the layout that it should not expand the text input. The `expand` property indicates the layout will not resize the presenter to take the whole available space.

```
TodoTaskPresenter >> updatePresenter
    task ifNotNil: [
        titlePresenter text: (task title ifNil: [ '' ])]
```

The method `updatePresenter` deserves a bit of explanation:

- indeed it is necessary to check that the task is not nil because the method is executed just after the instance creation and we did not initialize the task with a default task. The alternative to this ifNotNil: is to initialize the instance variable task to a kind of NullObject, here a null task.
- `titlePresenter text: (aTask title ifNil: [ '' ])` changes the contents of our text input presenter.

Now, we need to define this presenter to act as a dialog. And we do it in the same way (almost) we defined `TodoListPresenter` as window. But to define a *dialog presenter* we need to define the method `initializeDialogWindow:`.

```
TodoTaskPresenter >> initializeDialogWindow: aDialogWindowPresenter

    aDialogWindowPresenter
        title: 'New task';
        initialExtent: 350@120;
        addButton: 'Accept' do: [ :dialog |
            self accept.
            dialog close ];
        addButton: 'Cancel' do: [ :dialog |
            dialog close ]
```

Here, along with the already known `title:` and `initialExtent:` we added:

- `addButton: 'Save' do: [ ... ]`. This will add buttons to our dialog window. And you need to define its behavior (the `accept` method).
- `addButton: 'Cancel' do: [ ... ]`. This is the same as the positive action, but here we do not want to do anything, that's why we just send `dialog close`.

How would be the `accept` method? Very simple.

```
TodoTaskPresenter >>accept
    self task title: titlePresenter text; save
```

Tip: You can try your dialog even if not yet integrated to your application by executing

```
TodoTaskPresenter new
    task: TodoTask new;
    openModal.
```

1.7 Using the dialog

So, now we can use it. And to use it we need to define how we want to present that ”add task” option. It can be a toolbar, an actionbar or a simple button. To remain simple and expand a little what we already know about layouts, we will use a simple button and for that, we go back again to the method `TodoListPresenter >> initializePresenters`. We will add a new instance variable named `addButton` to the `TodoListPresenter` class.

```
TodoListPresenter >> initializePresenters

    | addButton |
    todoListPresenter := self newTable
        addColumn: ((SpCheckBoxTableColumn
                    evaluated: [:task | task isDone])
                width: 20;
                onActivation: [ :task | task done: true ];
                onDeactivation: [ :task | task done: false ];
                yourself);
        addColumn: (SpStringTableColumn
                    title: 'Title' evaluated: [:task | task title]);
        yourself.

    addButton := self newButton
        label: 'Add task';
        action: [ self addTask ];
        yourself.

    self layout: (SpBoxLayout newTopToBottom
        spacing: 5;
        add: todoListPresenter;
        add: (SpBoxLayout newLeftToRight
                addLast: addButton expand: false;
                yourself)
        expand: false;
        yourself)
```

What have we added here? First, we added a button.

- `newButton` creates a button presenter.
- `label:` sets the label of the button.
- `action:` sets the action block to execute when we press the button.

Second, we modify our layout by adding a new layout! **Yes, layouts can be composed!**.

- `spacing: 5` is used to set the space between presenters, otherwise they will apprear stick

together and this is not visually good.

- `SpBoxLayout newLeftToRight` creates a box layout that will be arranged horizontally.
- `addLast: addButton expand: false` adds the button at the end (sorting it effectively at the end, as if it would be a dialog). The `expand` property indicates the layout will not resize the button to take the whole available space.
- finally, the layout itself was added with `expand` set to `false` to prevent the button to take half the size vertically.

NOTE: The expand property is important to place your elements correctly, but it does not do magic: when this property is set the layout will take the default size of the added presenter to determine its place. But it may happen that the defaults are not good and there are different things you can do (like define your own style), but this will not covered in this tutorial (we will work on that later).

Finally we create the method `addTask`.

```
TodoListPresenter >> addTask

    (TodoTaskPresenter newApplication: self application)
        task: TodoTask new;
        openModal.
    self updatePresenter
```

What we did here?

- With `TodoTaskPresenter newApplication: self application` we create the dialog presenter but not by calling `new` as usual but `newApplication:` and passing the application of current presenter. This is **fundamentally important** to keep your dialogs chained as part of your application. If you skip this, what will happen is that the presenter will be created in the *default application* of Pharo, which is called `NullApplication`. You do not want that.
- `task:` set a new task.
- `openModal` will open the dialog in modal way. It means the execution of your program will be stop until you accept or cancel your dialog.
- `updatePresenter` will call the method we defined, to update your list.

Figure 1-4 shows how the task manager looks like.

1.8 Refactoring initializePresenters

In Spec you can either define a layout as we have done before using the message `layout:` or define a separate method named `defaultLayout` and this

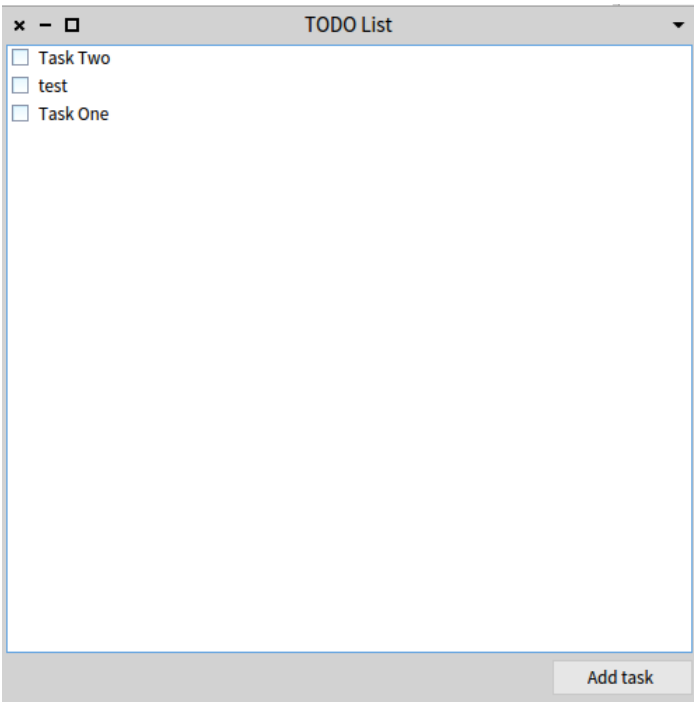

Figure 1-4 Task Manager with a button to add tasks.s

is what we are doing now to avoid to have too long method with different concerns.

```
TodoListPresenter >> defaultLayout

    ^ SpBoxLayout newTopToBottom
        spacing: 5;
        add: todoListPresenter;
        add: (SpBoxLayout newLeftToRight
                addLast: addButton expand: false;
                yourself)
        expand: false;
        yourself
```

Now the method `initialzePresenters` gets a single focus: the one of defining the subcomponents.

```
TodoListPresenter >> initializePresenters

    todoListPresenter := self newTable
        addColumn:
            ((SpCheckBoxTableColumn evaluated: [ :task |  task
```

```
    isDone ])
                width: 20;
                onActivation: [ :task | task done: true ];
                onDeactivation: [ :task | task done: false ];
                yourself);
        addColumn:
            (SpStringTableColumn
                title: 'Title'
                evaluated: [ :task | task title ]);
                yourself.
    addButton := self newButton
        label: 'Add task';
        action: [ self addTask ];
        yourself
```

1.9 Add edit and remove

But a task list is not just adding tasks. Sometimes we want to edit a task or even remove it. Let's add a context menu to table for this, and for it we will always need to modify `initializePresenters`.

```
TodoListPresenter >> initializePresenters

    todoListPresenter := self newTable
        addColumn:
            ((SpCheckBoxTableColumn evaluated: [ :task | task isDone
    ])
                width: 20;
                onActivation: [ :task | task done: true ];
                onDeactivation: [ :task | task done: false ];
                yourself);
        addColumn:
            (SpStringTableColumn
                title: 'Title'
                evaluated: [ :task | task title ]);
        yourself.
    todoListPresenter contextMenu: self todoListContextMenu.

    addButton := self newButton
        label: 'Add task';
        action: [ self addTask ];
        yourself
```

What is added now?

- `contextMenu: self todoListContextMenu` sets the context menu to what is defined in the method `todoListContextMenu`. Let us study right now.

```
TodoListPresenter >> todoListContextMenu

    ^ self newMenu
        addItem: [ :item | item
                    name: 'Edit...';
                    action: [ self editSelectedTask ] ];
        addItem: [ :item | item
                    name: 'Remove';
                    action: [ self removeSelectedTask ] ]
```

This method creates a menu to be displayed when pressing right-click on the table. Let's see what it contains:

- `self newMenu` as all other *factory methods*, this creates a menu presenter to be attached to another presenter.
- `addItem: [ :item | ... ]` add an item, with a `name:` and an associated `action:`

And now let's define the actions

```
TodoListPresenter >> editSelectedTask
    (TodoTaskPresenter newApplication: self application)
        task: todoListPresenter selection selectedItem;
        openModal.
    self updatePresenter
```

```
TodoListPresenter >> removeSelectedTask
    todoListPresenter selection selectedItem remove.
    self updatePresenter
```

As you see, `editSelectedTask` is almost equal to `addTask` but instead of adding a new task, it takes the selected task in our table by sending `TodoListPresenter selection selectedItem`. Remove simply takes the selected item and send the `remove` message.

1.10 Switching the backend to Gtk

Since we are developing a Spec2 application, we can decide to use Gtk as a backend instead of Morphic. How do we do this?

You need to load the Gtk backend as follows:

```
Metacello new
    repository: 'github://pharo-spec/mars-gtk';
    baseline: 'Mars';
    load.
```

(Do not worry if you have a couple of messages asking to "load" or "merge" a Spec2 package, this is because the baseline has Spec2 in its dependencies and it will "touch" the packages while loading the Gtk backend).

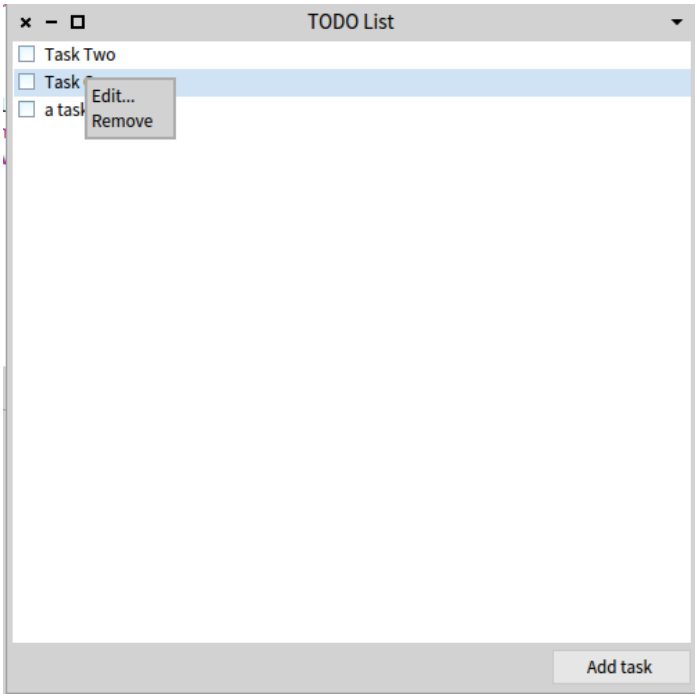

Figure 1-5 First full version of the Task List Manager.

Now, you can execute

```
TodoApplication new
    useBackend: #Gtk;
    run.
```

And that's all, you have your Todo application running as shown by Figure **??**.

1.11 Conclusion

In this tutorial we show that with Spec presenters are responsible of defining

- their subcomponents
- their layouts (how such components are displayed)
- how such components interact
- and the logic of the application (here we just added and removed elements from a list).

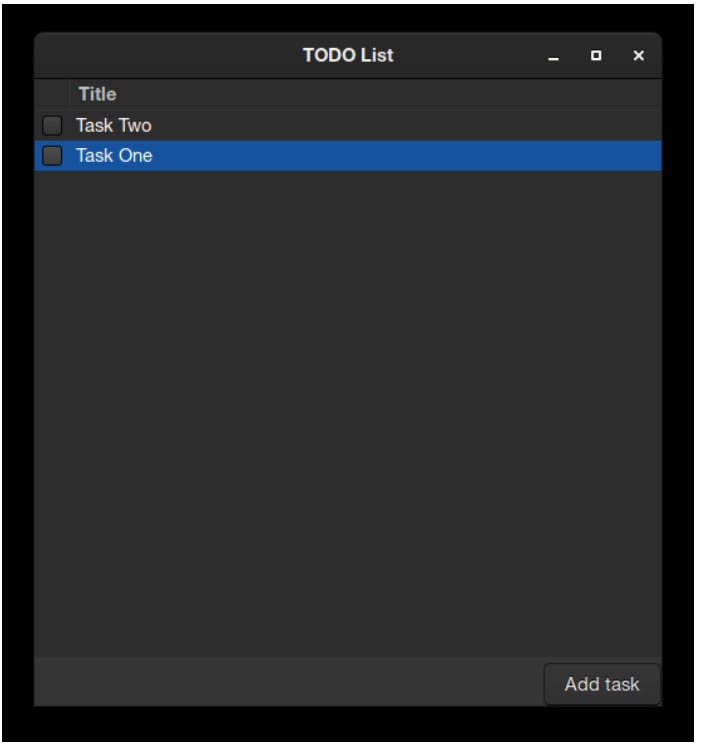

Figure 1-6 Task Manager running in GTK.