Commander20: a Command Framework

Stéphane Ducasse and Julien Delplanque

November 30, 2020

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Introduction

Commands are reification of actions. They can be used to implement actions in many contexts such as text-editors, debuggers, web-browsers, etc. Since recent versions, Pharo had no unified way to express commands and this led to multiple implementation of the command design pattern spreading across projects. Each project was re-implementing the design-pattern leading to slightly different version preventing the re-usability of common parts. To solve this problem we started to analyze existing solutions. We designed Commander20 framework that Pharo will use as canonical implementation of the command design pattern. With time, all custom-implementations of the command design pattern will be migrated to use Commander20. This booklet describes how commands are expressed in Pharo using Commander20.

To set some stage we start by defining a simple domain and applications in Spec20. Then we show how commands can be defined. In the final chapter we discuss the design of the framework with more details.

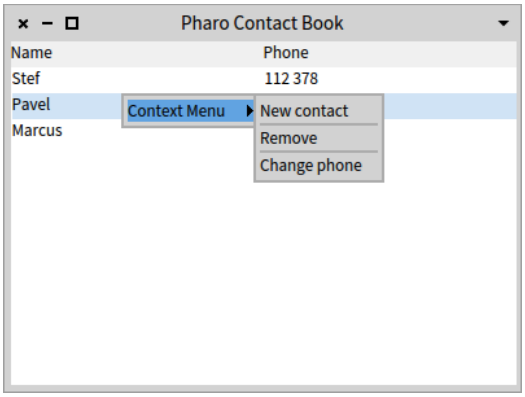
The code used of the contact book is available at https://github.com/Ducasse/EgContactBook. The code contains tests that are not listed in this booklet but we encourage you to read them.

CHAPTER 2

A Simple Contact Book

In this chapter we develop a simple model for a contact book. Then we define a user interface. This example will be used later in the book as an example to explain concepts such as commands, applications, windows.

Now it is more a replay of the concept previously mentioned. We start by implementing classes modeling the domain and then we will add a basic graphical user interface to obtain a little application as shown in Figure 2-1.


Pharo Contact Book

Name	Phone
Stef	112 378
Pavel	
Marcus	

Figure 2-1 A rudimentary contact book application.

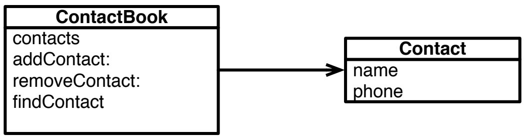


Figure 2-2 A simple model for the contact book.

2.1 Contact book model

The model for the domain of our example is composed of two classes: Contact and ContactBook as shown in Figure 2-2.

Contact

The class modeling a contact is defined as follow.

```
Object subclass: #EgContact
  instanceVariableNames: 'name phone'
  classVariableNames: ''
  package: 'EgContactBook'
```

It just defines a `printOn:` method and a couple of accessors.

```
EgContact >> printOn: aStream

    super printOn: aStream.
    aStream nextPut: $(.
    aStream nextPutAll: name.
    aStream nextPut: $).
```

```
EgContact >> name
  ^ name
```

```
EgContact >> phone
  ^ phone
```

```
EgContact >> name: aString
  name := aString
```

```
EgContact >> phone: anObject
  phone := anObject
```

```
EgContact >> hasMatchingText: aString
  ^ name includesSubstring: aString caseSensitive: false
```

```
EgContact class >> name: aNameString phone: aPhoneString
  ^ self new
    name: aNameString;
    phone: aPhoneString;
    yourself
```

ContactBook

Now we define the class modeling the contact book. As for the contact class, it is simple and quite straighforward.

```
Object subclass: #EgContactBook
  instanceVariableNames: 'contacts'
  classVariableNames: ''
  package: 'EgContactBook'
```

```
EgContactBook >> initialize
    super initialize.
    contacts := OrderedCollection new
```

```
EgContactBook >> contacts
  ^ contacts
```

```
EgContactBook >> contacts: aColl
  contacts := aColl
```

We add the possibility to add and remove a contact

```
EgContactBook >> addContact: aContact
  contacts add: aContact
```

```
EgContactBook >> removeContact: aContact
  contacts remove: aContact
```

```
EgContactBook >> addContact: newContact after: contactAfter
  contacts add: newContact after: contactAfter
```

We add a simple testing method in case one want to write some tests (which we urge you to do).

```
EgContactBook >> includesContact: aContact
  ^ contacts includes: aContact
```

And now we add a method to create a contact and add it to the contact book.

```
EgContactBook >> add: contactName phone: phone
  | contact |
  contact := EgContact new name: contactName; phone: phone.
  self addContact: contact.
  ^ contact
```

Finally some facilities to query the contact book.

```
EgContactBook >> findContactsWithText: aText
  ^ contacts select: [ :e | e hasMatchingText: aText ]
```

```
EgContactBook >> size
  ^ contacts size
```

Pre-filling up the contact book

Since we want to have some contacts and we way to keep them without resorting to a database or file we set some class instance variables.

We defined two class instance variables: `family` and `coworkers` and define some class method accessors as follows:

```
EgContactBook class >> family
  ^family ifNil: [
    family := self new
      add: 'John' phone: '342 345';
      add: 'Bill' phone: '123 678';
      add: 'Marry' phone: '789 567';
      yourself]
```

```
EgContactBook class >> coworkers
  ^coworkers ifNil: [
    coworkers := self new
      add: 'Stef' phone: '112 378';
      add: 'Pavel' phone: '898 678';
      add: 'Marcus' phone: '444 888';
      yourself]
```

We add one method to be able to reset them if necessary. The `<script>` pragma tells the system browser to add a small button to execute `reset` method easily.

```
EgContactBook class >> reset
  <script>
  coworkers := nil.
  family := nil
```

2.2 A simple graphical user interface

Now we define the graphical user interface (GUI) to expose the model to the user. The targeted GUI is shown in Figure 2-3.

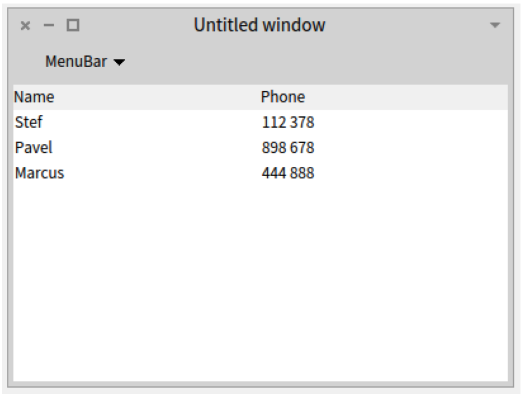
We define the class `EgContactBookPresenter`. It holds a reference to a contact book and it is structured around a table.

```
SpPresenter subclass: #EgContactBookPresenter
  instanceVariableNames: 'table contactBook'
  classVariableNames: ''
  package: 'EgContactBook'
```

We define an accessor for the contact book and the table.

```
EgContactBookPresenter >> contactBook
  ^ contactBook
```

```
EgContactBookPresenter >> table: anObject
  table := anObject
```



Name	Phone
Stef	112 378
Pavel	898 678
Marcus	444 888

Figure 2-3 A rudimentary contact book application.

```
EgContactBookPresenter >> table
  ^ table
```

Initializing the model

We specialize the method `setModelBeforeInitialization:` that is invoked by the framework to assign the `contactBook` instance variable to the object passed during the execution of the expression `(EgContactBookPresenter on: EgContactBook coworkers) openWithSpec`.

```
EgContactBookPresenter >> setModelBeforeInitialization: aContactBook
  super setModelBeforeInitialization: aContactBook.
  contactBook := aContactBook
```

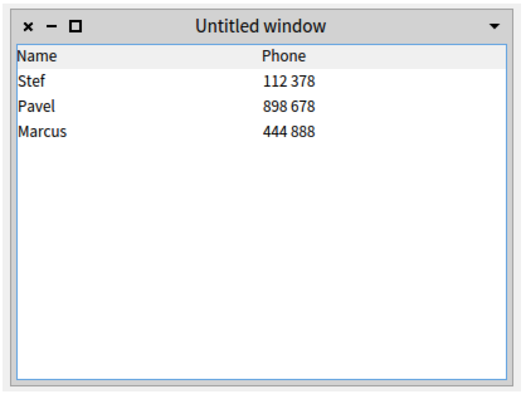
Layout

```
EgContactBookPresenter class >> defaultLayout
  ^ SpBoxLayout newTopToBottom add: #table; yourself
```

Widget initialization

We initialize the table to display two columns for the name and the phone. The respective accessor messages will be sent to the elements to fill up the columns. Finally the table contents is set using the contact book contents.

```
EgContactBookPresenter >> initializePresenters
  table := self newTable.
  table
    addColumn: (StringTableColumn title: 'Name' evaluated: #name);
    addColumn: (StringTableColumn title: 'Phone' evaluated: #phone).
  table items: contactBook contents.
```



Name	Phone
Stef	112 378
Pavel	898 678
Marcus	444 888

Figure 2-4 First version of the GUI without menus and toolbar.

Now we can start opening the UI by executing the following snippet `(EgContactBookPresenter on: EgContactBook coworkers) openWithSpec`

We define a class method to be able to easily re-execute the set up.

```
EgContactBookPresenter class >> coworkersExample
  <example>
  ^ (self on: EgContactBook coworkers) openWithSpec
```

You should obtain the following GUI as shown in Figure 3-1.

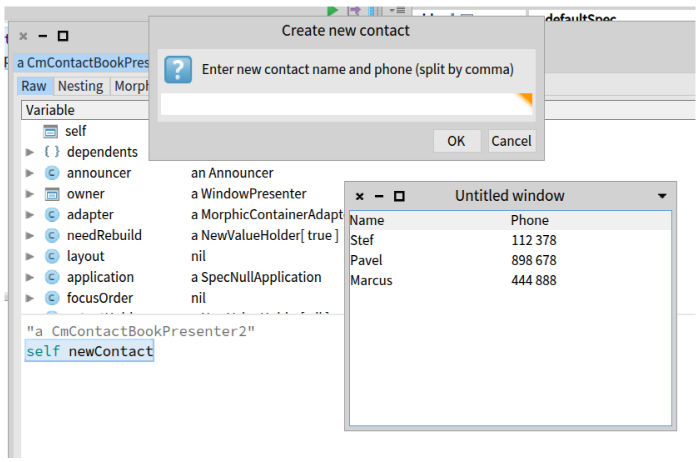
Interacting with user

We now implement the method that will open a window to ask the user to create a new contact for the contact book.

```
EgContactBookPresenter >> newContact
  | rawData splitted |
  rawData := self
    request: 'Enter new contact name and phone (split by comma)'
    initialAnswer: ''
    title: 'Create new contact'.
  splitted := rawData splitOn: $,.
  (splitted size = 2 and: [ splitted allSatisfy: #isNotEmpty ])
    ifFalse: [ SpInvalidUserInput signal: 'Please enter contact name
     and phone (split by comma)'  ].

  ^ EgContact new
    name: splitted first;
    phone: splitted second;
    yourself
```

To test it, we can get access to the presenter as follows



Enter new contact name and phone (split by comma)

Variable	
self	
dependents	
announcer	an Announcer
owner	a WindowPresenter
adapter	a MorphicContainerAdapt
needRebuild	a NewValueHolder[true]
layout	nil
application	a SpecNullApplication
focusOrder	nil

```
"a CmContactBookPresenter2"
self newContact
```

Name	Phone
Stef	112 378
Pavel	898 678
Marcus	444 888

Figure 2-5 Playing inside the inspector.

```
(EgContactBookPresenter on: EgContactBook coworkers)
  openWithSpec presenter inspect
```

and you can send the `newContact` message as shown in Figure 2-5.

Some extra methods

We will also define the methods `isContactSelected` and `selectedContact` to know if a contact is currently selected and to return it. It will help us later to add contact just after the currently selected contact.

```
EgContactBookPresenter >> isContactSelected
  ^ self table selectedItems isNotEmpty
```

```
EgContactBookPresenter >> selectedContact
  ^ table selection selectedItem
```

2.3 Conclusion

We have a little contact book manager now that we can use to explain other topics.

CHAPTER 3

Commander: a Powerful and Simple Command Framework

Commander was a library originally developed by Denis Kudriashov. Commander 2.0 is the second iteration of such a library. It has been designed and developed by Julien Delplanque and Stéphane Ducasse. Note that Commander 2.0 is not compatible with Commander but this is really easy to migrate from Commander to Commander 2.0. We describe Commander 2.0 in the context of Spec 2.0, the user interface building framework. From then on, when we mention Commander we refer to Commander 2.0. In addition we show how to extend Commander to other needs.

3.1 Commands

Commander models application actions as first class objects following the Command design pattern. With Commander, you express commands and use them to generate menus, toolbar but also to script an application from the command line (the associated decorator has not been yet developed).

Every action is implemented as a separate command class (subclass of `CmCommand`) with an `execute` method and all state required for execution. The superclass defines the context in which the command should be executed. Then the class `CmCommand` introduces name and description.

We will show later that for UI framework, we need more information such as an icon and shortcut description. In addition we will present how commands can be decorated with extra functionality in an extensible way.

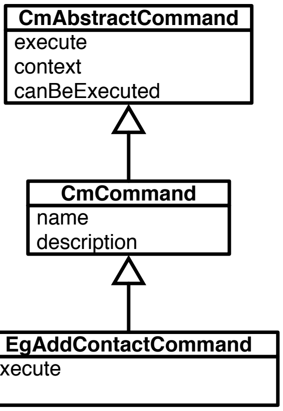


Figure 3-1 A simple command and its hierarchy.

A little example.

Here is a sketch of the logic of instantiating and executing a command.

```
(EgAddContactCommand new context: aPresenter) execute
```

We instantiate the command class and pass the presenter (in the case of a UI command) to which the command applies using the message `context:`. Note that a context can be an object, such as a presenter in the case of Spec but also be a block. Passing a block is interesting to give the command access to objects that depend on an execution that did not happen yet. And we send to the command the message `execute` to trigger its execution. This last point will often be done by the user interaction via toolbar or menuitems.

3.2 Defining commands

A command is a simple object instance of a subclass of the class `CmCommand`. It has a description, a name (this name can be either static or dynamic as we will shown later on). In addition, it has a context from which it extracts information to execute itself. In its basic form there is not much more than that.

Let us have a look at examples. We will define some commands for the ContactBook application and illustrate how they can be turned into menu and menubar.

Note that we will present how Commander supports Spec menu and menubar creations. However such functionalities are not in the core of Commander. We show them because first this is important to illustrate how to build user interfaces elements with Commander but also because such functionalities show that Commander can be extended in a way that end-users do not have to feel they are using special extensions. We will come back to such point in the last chapter of this book to show to potential extenders of Commander that they can get inspiration from the Spec extensions.

3.3 Adding some convenience methods

For convenience reasons, we define a common superclass named `EgContactBookCommand` to all the commands of the contact book application.

```
CmCommand subclass: #EgContactBookCommand
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'EgContactBook'
```

We define a simple helper method to make the code more readable

```
EgContactBookCommand >> contactBookPresenter
  ^ self context
```

For the same reason, we define another helper to access the contact book and the selected item.

```
EgContactBookCommand >> contactBook
  ^ self contactBookPresenter contactBook
```

```
EgContactBookCommand >> selectedContact
  ^ self contactBookPresenter selectedContact
```

Using such helper methods we define the method `hasSelectContract` as follows:

```
EgContactBookCommand >> hasSelectedContact
  ^ self contactBookPresenter isContactSelected
```

Adding the add contact command

We define a new subclass named `EgAddContactCommand` to define the command that represents the addition of a contact.

```
EgContactBookCommand subclass: #EgAddContactCommand
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'EgContactBook'
```

```
CmAddContactCommand >> initialize
  super initialize.
  self
    basicName: 'New contact';
    basicDescription: 'Creates a new contact and add it to the
     contact book.'
```

```
CmAddContactCommand >> execute
  | contact |
  contact := self contactBookPresenter newContact.
  self hasSelectedContact
    ifTrue: [ self contactBook
          addContact: contact
          after: self selectedContact ]
    ifFalse: [ self contactBook addContact: contact ].
  self contactBookPresenter updateView
```

We should define the method `updateView` to refresh the contents of the table.

```
EgContactBookPresenter >> updateView
  table items: contactBook contacts
```

Now in the inspect pane we can simply execute the command as follows:

```
(EgAddContactCommand new context: self) execute
```

Excuting the command should ask you to give a name and a phone number and will get added to the list.

We can also execute the following snippet.

```
| presenter cmd |
presenter := EgContactBookPresenter on: EgContactBook coworkers.
cmd := EgAddContactCommand new context: presenter.
cmd execute
```

3.4 Adding the remove contact command

We define now another command to remove a command. This example is interesting because it does not involve any UI interaction. It shows that a command is not necessarily linked to UI interaction.

```
EgContactBookCommand subclass: #EgRemoveContactCommand
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'EgContactBook'
```

```
EgRemoveContactCommand >> initialize
  super initialize.
  self
    name: 'Remove';
    description: 'Removes the selected contact from the contact
     book.'
```

This command definition illustrates how we can control when a command should or not be executed. The method `canBeExecuted` allows one to specify such condition.

```
EgRemoveContactCommand >> canBeExecuted
  ^ self context isContactSelected
```

The method `execute` is straightforward.

```
EgRemoveContactCommand >> execute
  self contactBook removeContact: self selectedContact.
  self contactBookPresenter updateView
```

The following test validates the correct execution of the command.

```
EgContactCommandTest >> testRemoveContact
  self assert: presenter contactBook size equals: 3.
  presenter table selectIndex: 1.
  (EgRemoveContactCommand new context: presenter) execute.
  self assert: presenter contactBook size equals: 2
```

3.5 Turning commands into menu items

Now that we have our commands we would like to reuse them and turn them into menus. In Spec, commands that are transformed into menu items are structured into a tree of command instances. The **class** method `buildCommandsGroupWith:forRoot:` of `SpPresenter` is a hook to let presenters populate the root of the command instance tree.

A command is transformed into a command for Spec using the message `forSpec`. We will show later that we can add UI specific information to a command such as an icon and a shortcut.

The method `buildCommandsGroupWith:forRoot:` registers commands to which the presenter instance is passed as context. Note that here we just add plain commands, but we can also create groups. This is also in this method that we will specify toolbar.

```
EgContactBookPresenter class >>
  buildCommandsGroupWith: presenter
  forRoot: rootCommandGroup

  rootCommandGroup
    register: (EgAddContactCommand forSpec context: presenter);
    register: (EgRemoveContactCommand forSpec context: presenter)
```

We have now have to attach the root of the command tree to the table. This is what what we do with the new line in the `initializePresenters` method. Notice that we have the full control and as we will show we could select a subpart of the tree (using the message `/`) and defining as root for given component.

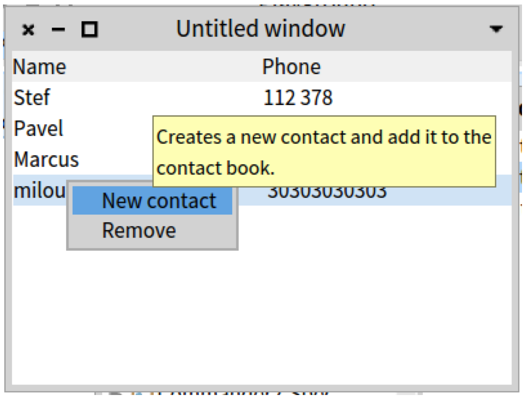


Figure 3-2 With two menu items with groups.

```
EgContactBookPresenter >> initializePresenters
  table := self newTable.
  table
    addColumn: (SpStringTableColumn title: 'Name' evaluated: #name);
    addColumn: (SpStringTableColumn title: 'Phone' evaluated:
    #phone).
  table contextMenu: [ self rootCommandsGroup beRoot asMenuPresenter
    ].
  table items: contactBook contacts.
```

Reopening the interface (`EgContactBookPresenter on: EgContactBook coworkers) openWithSpec` you should see the menu items as shown in Figure 3-2. As we will show later we could even replace a menu item by another one, changing its name, or icon in place.

3.6 Introducing groups

Commands can be managed in groups and such groups can be turned into corresponding menu item sections or submenus. The key hook method is the class method named `buildCommandsGroupWith: aPresenterInstance forRoot:`.

Here we give an example of such a grouping as menu section (`beDisplayedAsGroup`).

We create two methods creating each a simple group: one for adding and one for removing contracts. Each of the method could contain multiple commands. Here we just add one using the message `register:`.

```
EgContactBookPresenter class >> buildAddingGroupWith: presenter
  ^ (CmCommandGroup named: 'Adding') asSpecGroup
      description: 'Commands related to contact addition.';
      register: (EgAddContactCommand forSpec context: presenter);
      beDisplayedAsGroup;
      yourself
```

Note that the message `asSpecGroup` sent to a group.

```
EgContactBookPresenter class >> buildRemovingGroupWith: presenter
  ^ (CmCommandGroup named: 'Removing') asSpecGroup
      description: 'Commands related to contact removal.';
      register: (EgRemoveContactCommand forSpec context: presenter);
      beDisplayedAsGroup;
      yourself
```

We group the previously defined groups together under contextual menu for example.

```
EgContactBookPresenter class >> buildContextualMenuGroupWith:
    presenter
  ^ (CmCommandGroup named: 'Context Menu') asSpecGroup
      register: (self buildAddingGroupWith: presenter);
      register: (self buildRemovingGroupWith: presenter);
      yourself
```

Finally we revisit the hook `buildCommandsGroupWith:forRoot:` to register the last group to the root command group.

```
EgContactBookPresenter class >>
  buildCommandsGroupWith: presenter
  forRoot: rootCommandGroup

  rootCommandGroup
    register: (self buildContextualMenuGroupWith: presenter)
```

Reopening the interface `(EgContactBookPresenter on: EgContactBook coworkers) openWithSpec` you should see the menu items inside a `'Context Menu'` as shown in Figure 3-3.

To show you that we can also select a part of the command tree we select the `'Context Menu'` group and we declare it as the root of the table menu. In such case you will not see the `'Context Menu'` anymore.

```
EgContactBookPresenter >> initializePresenters

  table := self newTable.
  table
    addColumn: (SpStringTableColumn title: 'Name' evaluated: #name);
    addColumn: (SpStringTableColumn title: 'Phone' evaluated:
     #phone).
```

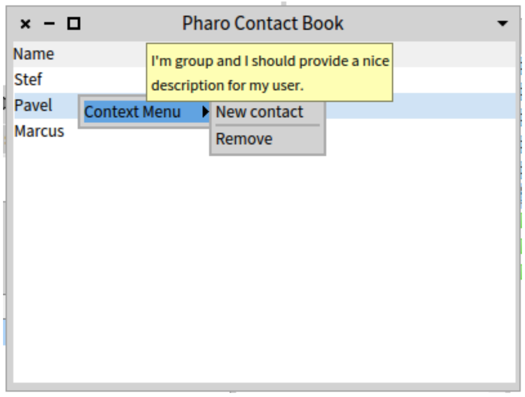


Figure 3-3 With a context menu.

```
  table contextMenu: [ (self rootCommandsGroup / 'Context Menu')
    beRoot asMenuPresenter ].
  table items: contactBook contacts
```

3.7 Extending menus

Building menu is nice, but sometimes we need to place an extra menu into an existing one, when we load a separate package. Commander supports this via a dedicated pragma, called `<extensionCommands>` that identifies extensions as we show it now.

Imagine that we have a new functionality that we want to add to the contact book and that this behavior is packaged in another package, here, `EgContactBook-Extensions`. First we will define a new command and second we will show how we can extend the existing menu to add a new menu item.

```
EgContactBookCommand subclass: #EgChangePhoneCommand
  instanceVariableNames: 'newPhone'
  classVariableNames: ''
  package: 'EgContactBook-Extensions'
```

```
EgChangePhoneCommand >> newPhone: anObject
  newPhone := anObject
```

```
EgChangePhoneCommand >> newPhone
  ^ newPhone
```

```
EgChangePhoneCommand >> initialize
  super initialize.
  self
    name: 'Change phone';
    description: 'Change the phone number of the contact.'
```

```
EgChangePhoneCommand >> execute
  self selectedContact phone: self contactBookPresenter newPhone.
  self contactBookPresenter updateView
```

We add ContactBookPresenter with the method newPhone the presenter to support the definition of the new phone number. The point here is not that this is method is or not packaged with the new command.

```
EgContactBookPresenter >> newPhone
  | phone |
  phone := self
    request: 'New phone for the contact'
    initialAnswer: self selectedContact phone
    title: 'Set new phone for contact'.
  (phone matchesRegex: '\d\d\d\s\d\d\d')
    ifFalse: [
      SpInvalidUserInput signal: 'The phone number is not well
    formated.
Should match "\d\d\d\s\d\d\d"' ].
  ^ phone
```

The last missing piece is the declaration of the extension. This one is done using the pragma <extensionCommands> on the class side of the presenter class as follows:

Here we see that using slash (/), we can select the group in which we want to add the item.

```
EgContactBookPresenter class >>
  changePhoneCommandWith: presenter
  forRootGroup: aRootCommandsGroup

  <extensionCommands>

  (aRootCommandsGroup / 'Context Menu')
    register: (EgChangePhoneCommand forSpec context: presenter)
```

3.8 Managing icons and shortcuts

By default a command does not know about Spec specific behavior, this is because a command does not have to be linked to UI. Now obviously you want to have icons and shortcut bindings when you are designing an interactive application.

Commander supports the addition of icons and shortcut key to commands. Let us see how it works from a user perspective. The framework offers two methods to set icon and shortcut key iconName: and shortcutKey: and we should specialize the method aSpecCommand as follows:

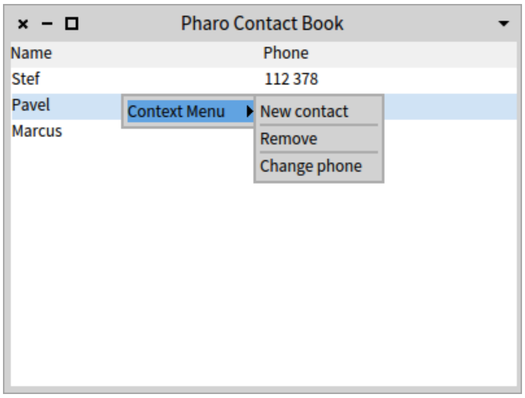


Figure 3-4 With menu extension.

```
EgRemoveContactCommand >> asSpecCommand
  ^ super asSpecCommand
    iconName: #removeIcon;
    shortcutKey: $x meta;
    yourself
```

```
EgRemoveContactCommand >> asSpecCommand
  ^ super asSpecCommand
    shortcutKey: $n meta;
    iconName: #changeAdd;
    yourself
```

Note that the commands are created using the message `forSpec` and this is this message that takes care about the calling of `asSpecCommand`.

3.9 Enabling shortcuts

To the time of this chapter writing, Commander management of shortcuts has not been pushed to Spec to avoid dependency to Commander. It is then the responsibility of your presenter to manage shortcuts as shown in the following method. Using the method `installShortcutsIn:`, we tell the command group to install the shortcut handler in the window.

```
EgContactBookPresenter >> initializeWindow: aWindowPresenter

  super initializeWindow: aWindowPresenter.
  self rootCommandsGroup installShortcutsIn: aWindowPresenter
```

3.10 In place customisation

Commander supports also the reuse and in place customisation of commands. It means that the instance representing a command can be reused and modified on the spot: for example its name or description can be adapted to the exact use context.

Here is an example that shows that we adapt twice the same command. We will

- define a basic command (inspect)
- instantiated it with two different contexts, contexts that are dynamic – we will pass blocks as contexts.
- change the description of the two commands.

Let us define a really simple and generic command which will simply inspect the object.

```
EgContactBookCommand subclass: #EgInspectCommand
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'EgContactBook-Extensions'
```

```
EgInspectCommand >> initialize
  super initialize.
  self
    name: 'Inspect';
    description: 'Inspect the context of this command.'
```

```
EgInspectCommand >> execute
  self context inspect
```

Using a block the context is computed at the moment the command is executed and the name and description can be adapted for its specific usage as shown in Figure 3-6.

```
EgContactBookPresenter class >>
  extraCommandsWith: presenter
  forRootGroup: aRootCommandsGroup

  <extensionCommands>

  aRootCommandsGroup / 'Context Menu'
    register:
      ((CmCommandGroup named: 'Extra') asSpecGroup
        description: 'Extra commands to help during development.';
        register:
          ((EgInspectCommand forSpec context: [ presenter
    selectedContact ])
            name: 'Inspect contact';
            description: 'Open an inspector on the selected
```

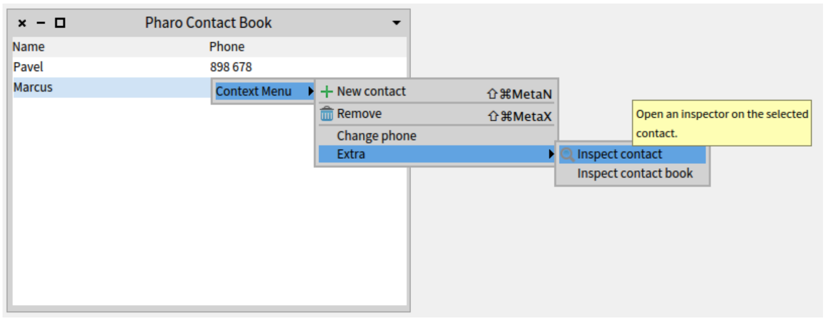


Figure 3-5 With menu extension.

```
    contact.';
            iconName: #smallFind;
            yourself);
        register:
          ((EgInspectCommand forSpec context: [ presenter
    contactBook ])
            name: 'Inspect contact book';
            description: 'Open an inspector on the contact book.';
            yourself);
        yourself)
```

3.11 Managing a menu bar

In addition to contextual menu creation sending the message `asMenuPresenter` to a group), commander supports menu bar and toolbar creation sending the message `asMenuBarPresenter` or `asToolbarPresenter` to a group. The logic is the same than for contextual menus : we define a group and register it under a given root and we specify to the presenter to use this group as a menubar.

Defining a print command.

Imagine that we have a new command to print the contact.

```
EgContactBookCommand subclass: #EgPrintContactCommand
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'EgContactBook'
```

```
EgPrintContactCommand >> initialize
  super initialize.
  self
    name: 'Print';
    description: 'Print the contact book in Transcript.'
```

```
EgPrintContactCommand >> execute

  Transcript open.
  self contactBook contacts do: [ :contact | self traceCr: contact
     name , ' - ' , contact name ]
```

Menu bar group.

We create a simple group that we call 'MenuBar' (but it could be called anything).

```
EgContactBookPresenter class >> buildMenuBarGroupWith: presenter
  ^ (CmCommandGroup named: 'MenuBar') asSpecGroup
    register: (EgPrintContactCommand forSpec context: presenter);
    yourself
```

We modify the root to get the menu bar group in addition the previous ones.

```
EgContactBookPresenter class >>
  buildCommandsGroupWith: presenter
  forRoot: rootCommandGroup

  rootCommandGroup
    register: (self buildMenuBarGroupWith: presenter);
    register: (self buildContextualMenuGroupWith: presenter)
```

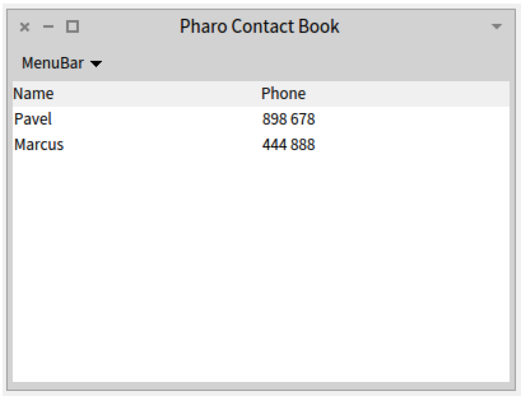
And we hook it into the widget as the last line of the `initializePresenters` method. Notice the use of the message `asMenuBarPresenter` and the addition of a new instance variable called `menuBar`. We have to do it to give the possibility to manage the menubar presenter as any other presenters.

```
EgContactBookPresenter >> initializePresenters
  table := self newTable.
  table
    addColumn: (SpStringTableColumn title: 'Name' evaluated: #name);
    addColumn: (SpStringTableColumn title: 'Phone' evaluated:
     #phone).
  table contextMenu: [ (self rootCommandsGroup / 'Context Menu')
     beRoot asMenuPresenter ].
  table items: contactBook contents.
  menuBar := (self rootCommandsGroup / 'MenuBar') asMenuBarPresenter.
```

Finally to get the menu bar you should declare it in the layout.

```
EgContactBookPresenter class >> defaultLayout

  ^ SpBoxLayout newTopToBottom
      add: #menuBar
      withConstraints: [ :constraints | constraints height: self
     toolbarHeight ];
```



Name	Phone
Pavel	898 678
Marcus	444 888

Figure 3-6 With menubar.

```
        add: #table;
        yourself
```

3.12 Conclusion

In this chapter we saw how you can define a simple command, execute it in a given context. We show how you can turn command into menu item in Spec20 by sending the message `forSpec`. You learned how we can reuse and customize commands. We presented groups of commands as a way to structure menus and menubars.

In the next chapter we will provide more details about certain UI aspects.

CHAPTER 4

Tips and Tricks for Spec

In this chapter we will detail some APIs that can be useful and some tips and tricks.

4.1 Icon Provider

The Commander' Spec extension (Commander for short in the rest of the chapter) has no preconcieved idea about where to look for icons. By default it uses internally the fonctionality provided by Spec presenters.

Now Commander lets you also specify your own source of icon provider using the message `iconProvider:`.

This way you can manage your own icon set without having to register in the global icon manager system.

4.2 PharoLauncher icon tricks

You may want to do a specific treatment on your icon form before displaying them. Here is a typical example made in the PharoLauncher to get a greyed icon. The command redefine the method `iconNamed:` and apply an effect before returning it.

```
PhLaunchImageCommand2 >> iconNamed: aName

  ^ (super iconNamed: aName) asGrayScaleWithAlpha
```

4.3 Extra Spec behavior

The integration of Commander into Spec20 allows one to access features that are only available for menu items. We can define the way a menu should be managed when it cannot be executed: it can either be hidden (message `beHiddenWhenCantBeRun`) or disabled (message `beDisabledWhenCantBeRun`). The last one is the default.

Therefore, when you specialize `canBeExecuted` to specify conditions under which a menu is executable, the command uses the specify strategy to show the fact that a menu item cannot be executed.

Similarly for there is the possibility to define where the command should be displayed this is particularly useful for toolbar with the message `beDisplayedOnRightSide` and `beDisplayedOnLeftSide`.

4.4 Contexts can be dynamic

If you want your command to work on a context that will change at execution, pass a block as argument of the `context:` message. The previous chapter showed such usage. We show the example in the following method:

```
EgContactBookPresenter class >>
  extraCommandsWith: presenter
  forRootGroup: aRootCommandsGroup

  <extensionCommands>

  aRootCommandsGroup / 'Context Menu'
    register:
      ((CmCommandGroup named: 'Extra') asSpecGroup
        description: 'Extra commands to help during development.';
        register:
          ((EgInspectCommand forSpec context: [ presenter
    selectedContact ])
            name: 'Inspect contact';
            description: 'Open an inspector on the selected
    contact.';
            iconName: #smallFind;
            yourself);
        register:
          ((EgInspectCommand forSpec context: [ presenter
    contactBook ])
            name: 'Inspect contact book';
            description: 'Open an inspector on the contact book.';
            yourself);
        yourself)
```

4.5 Toolbar

Commands can also be turned into a toolbar using the message `asToolbarPresenter` sent to a group of commands. If we take the same as the one presented for menu bar in the previous chapter.

We just have to use of the message `asToolbarPresenter`. We have to do it to give the possibility to manage the menubar presenter as any other presenters.

```
EgContactBookPresenter >> initializePresenters
  table := self newTable.
  table
    addColumn: (SpStringTableColumn title: 'Name' evaluated: #name);
    addColumn: (SpStringTableColumn title: 'Phone' evaluated:
     #phone).
  table contextMenu: [ (self rootCommandsGroup / 'Context Menu')
     beRoot asMenuPresenter ].
  table items: contactBook contents.
  menuBar := (self rootCommandsGroup / 'MenuBar') asToolbarPresenter.
```

Finally to get the menu bar you should declare it in the layout.

```
EgContactBookPresenter class >> defaultLayout

  ^ SpBoxLayout newTopToBottom
      add: #menuBar
      withConstraints: [ :constraints | constraints height: self
     toolbarHeight ];
      add: #table;
      yourself
```

An example taken from the Pharo tools, illustrates the use withi the method `asToolbarPresenterWith:`.

```
StPlayground >> buildToolbar

  ^ self toolbarActions
    asToolbarPresenterWith: [ :presenter |
      presenter
        displayMode: self application toolbarDisplayMode;
        addStyle: 'stToolbar' ]
```

```
StPlayground >> toolbarActions

  ^ CmCommandGroup forSpec
      register: (CmCommandGroup forSpec
        register: (StPlaygroundDoItCommand forSpecContext: self);
        register: (StPlaygroundPublishCommand forSpecContext: self);
        register: (StPlaygroundBindingsCommand forSpecContext: self);
        register: (StPlaygroundPagesCommand forSpecContext: self);
        yourself);
```

```
    yourself
```

4.6 Registration and navigation

Commands are often grouped together to act as menu groups. In the previous chapter we show that a group is structured as a composite tree of groups and commands. Adding elements to such composite is done via the messages `register: aGroupOrCommand`. You can register as many as commands or groups as you need as shown by the previous `toolbarActions` method

We already show that the message / navigates the tree and let you access the corresponding subtree.

The Spec extension supports also the notion of order and substitution as follows:

- `registerFirst: aGroupOrCommand` registers the argument as first.
- `registerLast: aGroupOrCommand` registers the argument as last
- `register: aGroupOrCommand after: another` and `register:aGroupOrCommand before: another` registers the argument relative to another element.
- `register: aGroupOrCommand insteadOf: another` replace an element by another.
- `unregister:` removes the element from the tree.

4.7 Conclusion

In this short chapter we saw some tips and tricks around Spec and Commander. The next chapter shows the design of commander and the extensibility mechanisms to that framework builder can simply reuse Commander instead of reinventing yet another command pattern framework.

CHAPTER 5

For framework designers

The design of Commander favors strong extensibility while keeping its use simple. The extensibility is brought in by using a simple decorator pattern. The idea is to propose a simple API to users while giving the possibility to modularly extend the framework. In this chapter, we explain the key aspects of Commander so that developers of other frameworks can use it as the root of their solution. We show the integration of Commander and Spec as a concrete use case.

5.1 Decorating commands

By default a command does not know about Spec specific behavior. It can be used in other contexts such as scripting libraries as this was the case with Gofer (Gofer was a scripting API to script monticello). The design of Commander supports the following scenario: the idea is that the core behavior of PharoLauncher should be able to be exposed as Clap command-line using the fact that PharoLauncher define commands.

The Spec project extends Commander so that we use commands with specific aspects related to Spec. It uses the fact that Commander allows us to decorate commands with decoration that are polymorphic to commands as shown in Figure 5-1.

Let us describe Figure 5-1:

- The common abstract root class `CmAbstractCommand` defines way to manage the context and execute a command.
- The class `CmBlockCommand` is a generic command whose behavior is specified using a block.

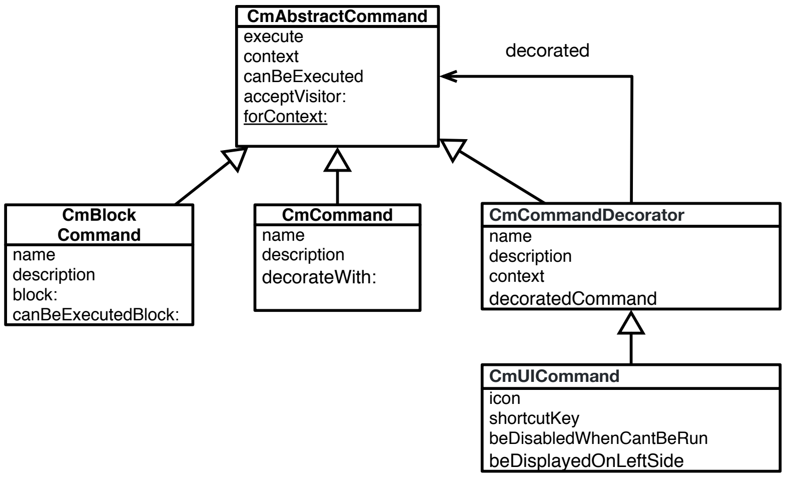


Figure 5-1 Commands and command decorators.

- The class `CmCommand` is the main root class of commands. Users will usually subclass it. For example, `EgAddContactCommand` subclasses it as shown in Figure 5-2.

Without Spec integration, the class `CmCommand` does not have any behavior related to Spec. The only interesting extension point is the `decorateWith:` method that is an extension hook as we show later.

- The class `CmCommandDecorator` is the root of command decorators. It merely delegates to a decorated command. It supports dynamic name and descriptions (to be able to get more specialized and dynamically updated commands). At runtime, decorators will be created to wrap a command and as such provide more information.
- The class `CmUICommand` is a decorator of command dedicated for UI related state and actions. Note that it is not linked per se to Spec. It offers some general functionalities related to UI, as we saw in previous chapters:
  - State: icon, shortcutKey
  - Behavior: defining icon, and shorcutKey

Now we are ready to study the Spec integration.

5.2 Modular Spec command decoration

Figure 5-2 shows how Commander is extended to support Spec specific behavior and this in a modular way. First the class `SpCommand` is a Spec specific

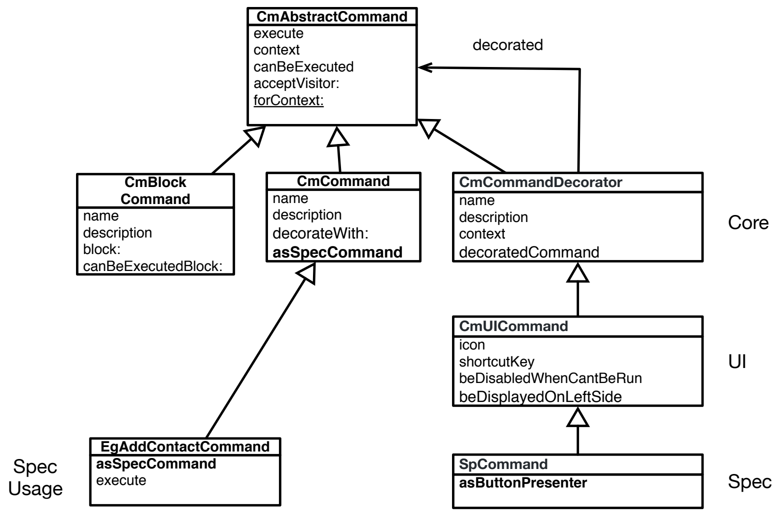


Figure 5-2 Spec decorations and use.

decorator. For example, it contains logic how to turn a command into a button presenter. Second, the package containing the Spec related code extends the class `CmCommand` with the `asSpecCommand` method. The method `asSpecCommand` decorates a command to define extra behavior responsible for the addition of ui related functionality and state.

The implementation is the following one

```
CmCommand >> asSpecCommand
  ^ self decorateWith: SpCommand
```

It means that the base command will be decorated by an instance of the class `SpCommand`.

Remember that the method `asSpecCommand` is directly or indirectly used by the developer to build commands that he will register to the command root of its presenter. The following method shows a typicall command instantiation.

```
EgContactBookPresenter >> buildAddingGroupWith: presenter
  ^ (CmCommandGroup named: 'Adding') asSpecGroup
      description: 'Commands related to contact addition.';
      register: (EgAddContactCommand forSpec context: presenter);
      beDisplayedAsGroup;
      yourself
```

The method `forSpec` is a handy creation class method.

```
CmCommand class >> forSpec
  ^ self new asSpecCommand
```

5.3 Spec Commander user perspective

What is interesting to note is that as an end-user the developer defining the command has just to define the method `asSpecCommand` without having to worry about the details of the implementation. Here is the command definition for the command of adding a contact in our contact book application.

```
EgAddContactCommand >> asSpecCommand
  ^ super asSpecCommand
    shortcutKey: $n meta;
    iconName: #changeAdd;
    yourself
```

The command `EgAddContactCommand` is a subclass of `CmCommand` and not `SpCommand`. If the execute method does not perform or use UI specific behavior this makes sense and it makes possible to use the command without such constraints. Now if your commands are UI specific you can also define them as subclass of `SpCommand`.

`SpCommand` is a decorator so inheriting from it would be a mistake because the state and behavior of command would not be available since there would be no decorated command.

5.4 Decorating group of commands

Commander uses the same logic for group of commands as shown by Figure 5-3. Let us describe the hierarchy.

- The class `CmAbstractCommandGroup` is an abstract root defining elementary operation of group.
- The class `CmCommandGroup` is the central class from an end-user point of view.
- The class `CmCommandGroupDecorator` is just a decorator.

On top of this the class `CmUICommandGroup` extends the basic decorator with ui related behavior as shown in Figure 5-4.

5.5 Spec decoration

The Spec extension then is based on the definition of a specific decorator `SpCommandGroup` and the method `asSpecGroup` defined as follows:

```
CmCommandGroup >> asSpecGroup
  ^ self decorateWith: SpCommandGroup
```

The class `SpCommandGroup` defines methods that are producing Spec object often using the

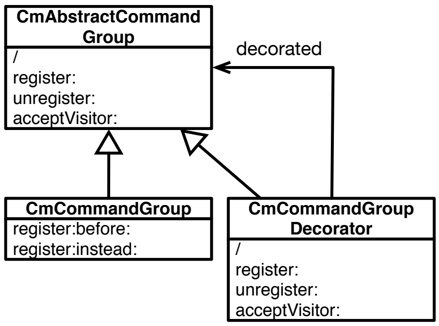


Figure 5-3 Group and group decorators.

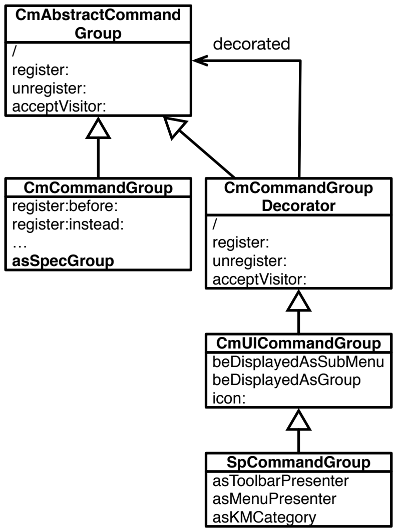


Figure 5-4 Spec's group and group decorators.

```
SpCommandGroup >> asMenuPresenter
  ^ SpMenuPresenterBuilder new
    visit: self;
    menuPresenter
```

5.6 Example of Visitor: toolbarBuilder

Commander defines a simple visitor. This visitor is used in many places. Here we present the `SpToolBarPresenterBuilder` responsible for turning groups into a toolbar.

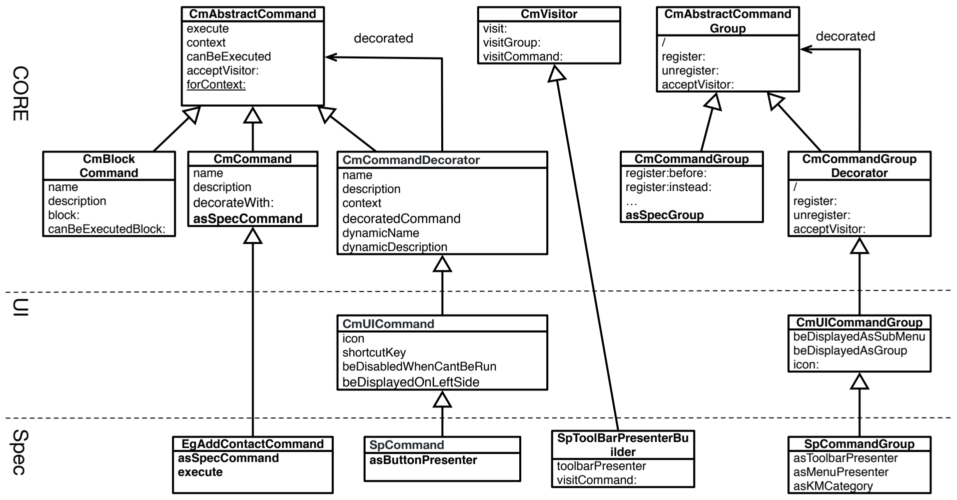


Figure 5-5 Full design: Core, UI support and Spec integration.

```
SpCommandGroup >> asToolbarPresenter
  ^ SpToolBarPresenterBuilder new
    visit: self;
    toolbarPresenter
```

```
SpvisitCommand: aCmCommandEntry
  aCmCommandEntry positionStrategy
    addButton: (SpToolBarButton new
            label: aCmCommandEntry name;
            help: aCmCommandEntry description;
            icon: aCmCommandEntry icon;
            action: [ aCmCommandEntry execute ];
            yourself)
    toToolbar: self toolbarPresenter
```

5.7 Conclusion

To conclude this chapter Figure 5-5 gives a full overview of the design and layers supported by Commander. We show that Commander proposes a simple way for user to express commands while at the same time commands can be modularly decorated to add extra behavior. Commander20 is a central piece of Pharo infrastructure and all the other commands solutions will be ported to it.

Bibliography