CHAPTER 1

Commander: A command pattern library

In this chapter you will learn about a new library to manage commands. The library is based on three main objects: *commands* that represent the actions to be executed, *command activators* that represent ways commnads are activated (i.e., using shortcuts, menus..) and *contexts* which represent selection of commands.

1.1 Commands

Commander models application actions as first class objects.

Every action is implemented as separate command class (subclass of `CmdCommand`) with `execute` method and all state required for execution.

An example

The `RenamePackageCommand` is defined as follows

```
SD:Denis please add it
```

1.2 Commands, activators and contexts

The commander library is structured around three main entities:

- *commands* that represent the actions to be executed,

- *command activators* that represent ways commands are activated (i.e., using shortcuts, menus..) and
- *contexts* which represent selection of commands.

To do SD:here we should have a diagram

Activators

Commands are reusable objects and applications provide various ways to access them: shortcuts, context menu, buttons, etc.. This information is attached to command classes as activator objects. Currently there are three types of activators:

- `CmdShortcutCommandActivator`
- `CmdContextMenuCommandActivator`
- `CmdDragAndDropCommandActivator`

Activators are declared in command class side methods tagged with pragma `<commandActivator>`. For example, the following method will allow `RenamePackageCommand` to be executed by shortcut in a given system browser:

```
RenamePackageCommand class >> packageBrowserShortcutActivator
  <commandActivator>
  ^CmdShortcutCommandActivator by: $r meta for: PackageBrowserContext
```

Similarly the following method `packageBrowserMenuActivator` define that the `RenamePackageCommand` can be executed from a context menu. Note the use of `CmdContextMenuCommandActivator` in the method, while the previous one was using `CmdShortcutCommandActivator`.

```
RenamePackageCommand class >> packageBrowserMenuActivator
  <commandActivator>
  ^CmdContextMenuCommandActivator byRootGroupItemFor:
    PackageBrowserContext
```

Contexts

Activators are always declared with the application context where they can be applied (`PackageBrowserContext` in example). An application should provide such contexts as subclasses of `CmdToolContext` with information related to the application state.

Every widget can bring its own context to interact with application as separate tool. For example, a system browser shows multiple panes which provide package, class and method contexts. Depending on the context, the browser will show different menus and provides different shortcuts.

To support activators, command should implement several methods: `What is a standardContext:?`

canBeExecutedInContext: aToolContext

The method `canBeExecutedInContext:` defines whether the command that be executed in a context. By default it returns true. But usually commands query context for specific information.

For example `RenamePackageCommand` requires a package and it defines this method as follows:

```
RenamePackageCommand >> canBeExecutedInContext: aToolContext
   ^aToolContext isPackageSelected
```

prepareFullExecutionInContext: aToolContext

With the method `prepareFullExecutionInContext:`, the command should retrieve all state required for execution. It can also ask user for extra data. For example `RenamePackageCommand` retrieves the package to be renamed from the context and asks the user for new name:

```
RenamePackageCommand >> prepareFullExecutionInContext: aToolContext


    package := aToolContext selectedPackage.
    newName := UIManager default
        request: 'New name of the package'
        initialAnswer: package name
        title: 'Rename a package'.
    newName isEmptyOrNil | (newName = package name)
    ifTrue: [ ^ CmdCommandAborted signal ]
```

Note that to abort command execution, a command can raise the `CmdCommandAborted` signal.

applyResultInContext: aToolContext

Purpose of the method `applyResultInContext:` is to be able interact with application when command completes. `I do not like the name of the method` why not `afterExecution or postActionInContext:`.

For example, when a user creates new package from a browser, we want to tool to show the package. We expressed that once the command is executed, the browser should open created package as follows:

```
CreatePackageCommand >> applyResultInContext: aToolContext
    aToolContext showPackage: resultPackage
```

`SD:hy this logic is defined in the command and not in the tool logic??`

Commands are supposed to be reusable for different contexts and these methods should be implemented with that in mind. They should not leak internal structure of contexts.

Specific contexts

Specific context can override activation methods and send its own set of messages to command.

For example:

```
SpecialContextA >> allowsExecutionOf: aCommand
   ^aCommand canBeExecutedInSpecialContextA: self

SpecialContextA >> prepareFullExecutionOf: aCommand
  aCommand prepareFullExecutionInSpecialContextA: self

SpecialContextA >> applyResultOf: aCommand
  aCommand applyResultInSpecialContextA: self
```

`I'm lost: where these methods come from? are they defined?`

By default `CmdCommand` can implement with standard context methods. And only particular commands will override them specifically:

```
CmdCommand >> prepareFullExecutionInSpecialContextA: aSpecialContextA
  self prepareFullExecutionInContext: aSpecialContextA

SomeCommand >> prepareFullExecutionInSpecialContextA:
    aSpecialContextA
  "special logic to prepare command for execution"
```

The way how concrete type of activator hooks into application is responsibility of application. Look at related sections for details on concrete activator.

In future Commander will provide deep integration with UI. And many things will work automatically.

1.3 Command execution

Context instances are used to perform command lookup. For example following expression will enumerate all shortcut activators relevant to package pane of a possible browser:

```
CmdShortcutCommandActivator
  allDeclaredFor: aPackageBrowserContext
  do: [ :declaredActivator | show all the code here please ]
```

A declared activator is in fact not ready for execution because it should be bound to a context. It should create new activator instance for this:

```
readyActivator := declaredActivator newActivationFor:
    aPackageBrowserContext
```

Ready activator is bound to given context and it keeps a new command instance. To execute command evaluate:

```
readyActivator executeCommand
```

Before execution, users should check that it is possible:

```
activator canExecuteCommandInContext: aPackageBrowserContext
```

There is a convenient method to enumerate only executable commands. Here is for example how to get all the executable commands for package related commands.

```
CmdShortcutCommandActivator
   allExecutableIn: aPackageBrowserContext
   do: [ :readyActivator |
   please show code that we can execute ]
```

1.4 Abstract menu activation

`SD:why abstract??.`

There are a lot of different types of menu: context menu, toolbar, morphic halo menu, etc. All of them are very similar: they show set of items to the user and when user selects one the action associated with item is evaluated. The main difference between them is the way how they are represented to user and where they are shown in application.

The commander library allows the building of different kind of menus based on commands and first class groups. Concrete type of menu is represented by concrete type of activator, subclass of `CmdMenuCommandActivator`.

These activators mark commands to be part of menu:

```
YourCommand >> yourAppMenuActivator
  <commandActivator>
  ^ConcreteMenuCommandActivator byItemOf: YourAppChildMenuGroup for:
    YourAppContext
```

Activator provides information about name, group and position of command inside menu:

- `menuItemName` (by default it is retrieved from command instance by sending the message `defaultMenuItemName`).
- `menuItemOrder` (higher value pushes command to the end of menu)
- `menuGroup` (root by default)

To set up these properties there are few instance creation methods:

- `byRootGroupItemFor: YourAppContext`
- `byRootGroupItemOrder: aNumber for: YourAppContext`
- `byItemOf: menuGroupClass for: YourAppContext`

- `byItemOf: menuGroupClass order: aNumber for: YourAppContext`

`CmdMenu` represents abstract tree structure for concrete menu implementation: `SD:I do not get how cmdMenu is related to ConcreteMenuCommandActivator` `SD: give a real example`

```
menu := CmdMenu activatedBy: aCommandActivatorClass
```

`I'm lost why do we need a commandActivatorClass here?`

It builds commands and group items using information from activators declared for given context:

```
menu buildInContext: aToolContext
```

This method can be called multiple times for different contexts. It allows one to build single menu for multiple parts of application. For example toolbar menu can include commands for all visible widgets.

Menu groups are represented by subclasses of `CmdMenuGroup`. They are used as classes to declare activators. Instances are only created during menu building.

Groups are containers of command items and other groups. They define following methods to describe menu structure:

- `parentGroup` on class side. By default it is CmdRootMenuGroup. Subclasses override it to define deep tree structure.
- `order` on instance side. Larger values pushed group to the end of menu
- `name` on instance side. It should be shown to user in concrete menu implementation.

Concrete menu activators extend menu objects and commands with specific methods to build concrete view elements. For example look at next section.

`SD: but next section refers to this one`

1.5 Context menu command activation

To add commands into context menu use `CmdContextMenuCommandActivator` (more details at 1.4). For example:

```
RenamePackageCommand class >> packageBrowserMenuActivator
  <commandActivator>
  ^CmdContextMenuCommandActivator byRootGroupItemFor:
    PackageBrowserContext
```

To build context menu morph use following expression:

```
menu := CmdContextMenuCommandActivator buildMenuFor: anAppMorph
    inContext: aToolContext
```

Context menu extends command with menu building method:

```
CmdCommand >> fillContextMenu: aMenu using: anActivator
```

It creates menu item morph and allows subclasses to define default label and icon:

- `defaultMenuItemName`
- `setUpIconForMenuItem: aMenuItemMorph`

Subclasses can override building method `fillContextMenu:using:` to represent themselves differently. For example they can create menu item with check box.

1.6 Shortcut command activation

To mark commands with shortcuts use CmdShortcutCommandActivator. For example:

```
RenamePackageCommand class >> packageBrowserShortcutActivator
  <commandActivator>
  ^CmdShortcutCommandActivator by: $r meta for: PackageBrowserContext
```

There are extra instance creation methods for standard shortcuts:

- `renamingFor: aToolContext`
- `removalFor: aToolContext`

To support shortcuts based on commands application should define specific kmDispatcher for target morphs:

```
YourAppMorph >> kmDispatcher
  ^CmdKMDispatcher attachedTo: self
```

with supporting method:

```
YourAppMorph >> createCommandContext
  ^YourAppContext for: self
```

1.7 Drag and drop command activation

Drag and drop command activation is different than others. To prepare command it needs two contexts. First context describes the place where drag was started. Second context describes drop target tool. So activator prepares command execution in both contexts. They bring different information. And together they are supposed to provide all required data for command execution without extra user requests.

According to this logic commands should define following supporting methods when they want drag and drop:

- `prepareExecutionInDragContext: aToolContext`
- `prepareExecutionInDropContext: aToolContext`
- `canBeExecutedInDropContext: aToolContext`

Also specific contexts can define own set of activation methods:

```
SpecialContextA >> allowsDropExecutionOf: aCommand
  ^aCommand canBeExecutedInSpecialDropContextA: self

SpecialContextA >> prepareDragActivationOf: aCommand
  aCommand prepareExecutionInSpecialDragContextA: self

SpecialContextA >> prepareDropExecutionOf: aCommand
  aCommand prepareExecutionInSpecialDropContextA: self
```

To activate commands with drag and drop use `CmdDragAndDropCommandActivator`. For example:

```
MoveMethodToClassCommand class >> methodBrowserDragAndDropActivator
  <commandActivator>
  ^CmdDragAndDropCommandActivator for: MethodBrowserContext
    toDropIn: ClassBrowserContext
```

And supporting methods would be:

```
MoveMethodToClassCommand >> prepareExecutionInDragContext:
    aToolContext
  super prepareExecutionInDragContext: aToolContext.
  methods := aToolContext selectedMethods

MoveMethodToClassCommand >> prepareExecutionInDropContext:
    aToolContext
    super prepareExecutionInDropContext: aToolContext.
    targetClass := aToolContext selectedClass
```