GUI application with Cuis-Smalltalk

Hilaire Fernandes

Department of Public Instruction
Geneva

November 2023

About me

- Educator in public school, Geneva, B.Math, Ma.Ed
- Computer scientist, Ma.CS, PhD.CS
- Free software enthusiast and user since 1998
- And of course, Smalltalk user since 2002

Contents

Why writing a GUI application with **Smalltalk**?

- Portable application
- Rapid prototyping
- Explore concepts and new ideas
- Learn from the environment itself
- Culture of design patterns
- Culture of testing
- Moldable environment
- Bugs hunt are fun, and fixes too

Why writing a GUI application with **Cuis**-Smalltalk?

- Libre Smalltalk, the three fundamental freedoms
  $\Rightarrow$ deploy & share as you want
- A Smalltalk system for humble human being
  $\Rightarrow$ system complexity under control, not on your way
- Cuis makes Morph correct
  $\Rightarrow$ hierarchy, coordinates system, floating point precision
- State of the art vector graphics engine
  $\Rightarrow$ make your app beautifully
- Develop your own vector widget
  $\Rightarrow$ with SVG icons

A guide for the beginners

These workshop and related documents:

- A step by step guide to get started elaborating a Cuis App
- How to organize your code
- Expose the different parts needed for a Cuis App,
- Which packages to rely on
- Which design patterns you may want to use
- *How to write your own vector widget* $\Rightarrow$ sorry not this time
- Which resources to use

An image viewer

An application with a toolbar to select an image and to operate transformations with undo/redo commands.

From scratch

The free operating system GNU/Linux is used along some Bash commands and scripts. You also need the `git` tool to access the Cuis repositories.
⇒ In your disk, create a new folder "`myProject`" for your project:

```
mkdir myProject
```

This is the place where you will pull the needed Cuis repositories, the virtual machine and the repository dedicated to your Cuis App.

Cuis image

We need to clone the Cuis official repo to get the latest image and updates along additional repositories of libraries we want to use. We suggest you read latter the official guides to keep this repository updated[5].

```
cd myProject
git clone --depth 1 \
   https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev.git
cd Cuis-Smalltalk-Dev
./clonePackageRepos.sh
```

Virtual Machine

Get the latest virtual machine for GNU/Linux[5]:

```
cd myProject
rm -r cogspur
wget -O cogspur.tgz \
   https://github.com/OpenSmalltalk/opensmalltalk-vm/\
   releases/latest/download/squeak.cog.spur_linux64x64.tar.gz
tar -zxvf cogspur.tgz
mv ./sqcogspur64linuxht ./cogspur
```

⇒ Test you can start Cuis:

```
../cogspur/squeak Cuis6.0-6xxx
```

Additional repositories

We want to install the SVG & Cuis-Smalltalk-UI repositories:

```
cd myProject
git clone --depth 1 https://github.com/Cuis-Smalltalk/\
   Cuis-Smalltalk-UI.git
git clone --depth 1 https://github.com/Cuis-Smalltalk/\
   SVG.git
git clone --depth 1 https://github.com/Cuis-Smalltalk/\
   Numerics.git
```

It installs the repositories `Cuis-Smalltalk-UI` and `SVG` as folders in `myProject/`. They contains package files `.pck.st`.

CuisApp

Cuis is repository agnostic. Use whatever suits you to track your Cuis app. For Dr. Geo I use Bazaar/Launchpad because it offers a web tool to edit translations in native languages[2].

⇒ For now, we just create an empty folder "`CuisApp`" in the Cuis-Smalltalk-Dev folder (and local copy of the repository):

```
cd myProject/Cuis-Smalltalk-Dev
mkdir CuisApp
```

⇒ All the source codes and related resources needed for our Cuis application will go there.

Folders

We need places for the source code, scripts, resources, i18n(Internationalization), build:

```
cd myProject/Cuis-Smalltalk-Dev/CuisApp
mkdir src i18n build
mkdir -p resources/graphics/icons resources/doc
```

In a Cuis workspace, you request for a package as a Feature:

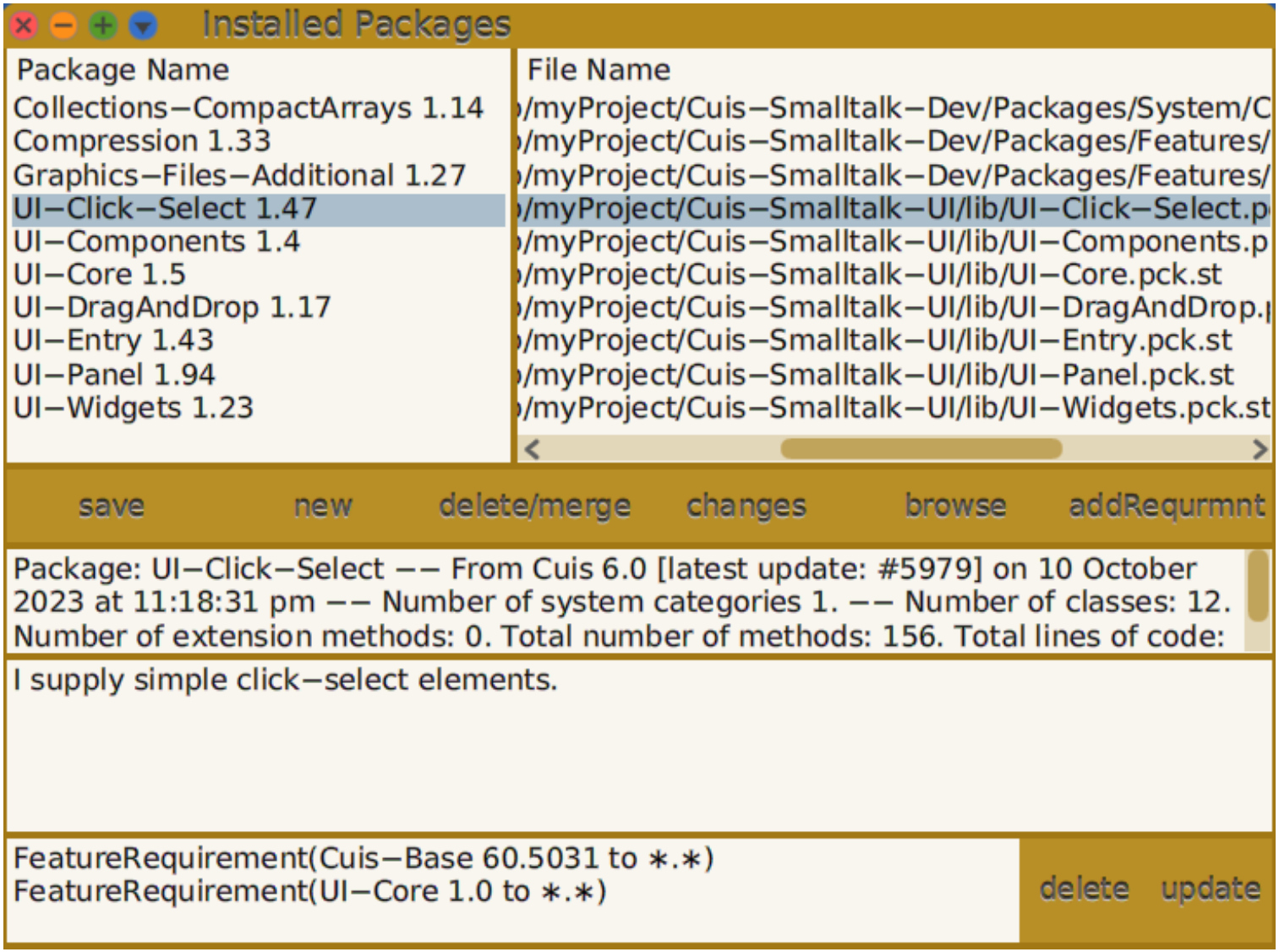
```
Feature require: 'UI-Components'
```

This installs the UI-Components package and more. It comes from the `Cuis-Smalltalk-UI/lib` repository.

In fact it is a meta-package, read it:

```
!provides: 'UI-Components' 1 4!
!requires: 'Cuis-Base' 60 5032 nil!
!requires: 'UI-DragAndDrop' 1 0 nil!
!requires: 'UI-Entry' 1 3 nil!
!requires: 'UI-Click-Select' 1 1 nil!
!requires: 'UI-Panel' 1 5 nil!
!requires: 'UI-Widgets' 1 0 nil!
```

Open it from the menu in the world World-Open-Installed Packages:



Our CuisApp package

To manage the code of our CuisApp, we create a package "CuisApp" and add the dependent packages we need:

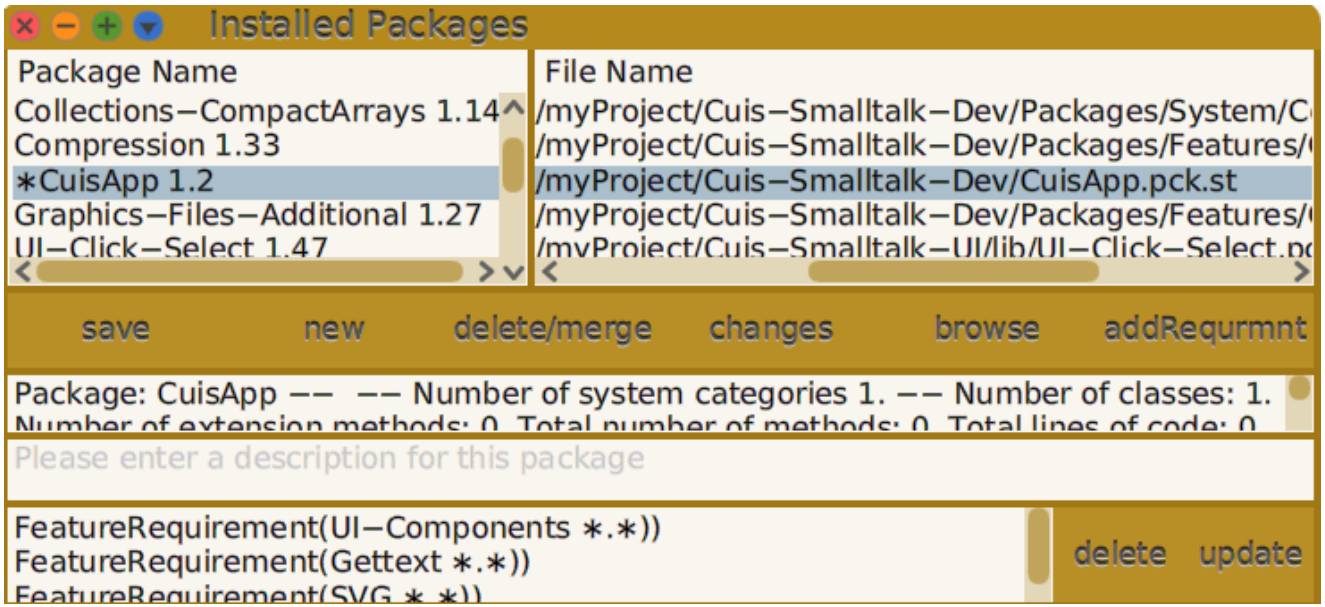
```
| package |
package := CodePackage
    named: 'CuisApp'
    createIfAbsent: true
    registerIfNew: true.
package featureSpec
    requires: (FeatureRequirement name: 'SVG');
    requires: (FeatureRequirement name: 'UI-Components');
    requires: (FeatureRequirement name: 'Goodies');
    requires: (FeatureRequirement name: 'Gettext').
package save
```

⇒ The package is saved along the Cuis image, in Cuis-Smalltalk-Dev.

Package content

In the class browser, we create a CuisApp class in the category **CuisApp**.
⇒ All classes in the categories **CuisApp** and **CuisApp-###** are automatically part of the CuisApp package.

Observe in the package manager the dependencies and the * indicating there are changes in the CuisApp package:



Test our package:

- From the package manager, save once more the CuisApp package, then move its file `CuisApp.pck.st` in `CuisApp/src`, in the CuisApp repository.
- Quit Cuis without saving
- Restart Cuis and in a workspace execute:

```
Feature require: 'CuisApp'
```

Our CuisApp package and its dependencies are installed at once. However our working environment is still a mess. We will address that.

Objectives

There are good habits to have when starting a new Cuis development session. To keep it up-front, we better automatise the involved processes:

- Use a fresh Cuis image at each start up
- Keep the default fresh Cuis image untouched
- Clean up from previous development session
- Install packages
- Adjust the Cuis environment to our taste

Run script

Clean-up from previous session, start Cuis with a copied fresh image and Smalltalk set up script:

```
#!/bin/bash
# Start CuisApp IDE
#
# Cuis Version release
release=‘ls Cuis6.0-????.image | cut -d - -f 2 | \
   cut -d . -f 1‘
cuis=Cuis6.0-$release
ide=CuisAppIDE
# Clean up from previous session
rm $ide.image $ide.changes $ide.user.* *.log
# Create fresh image for next session
cp $cuis.image $ide.image
cp $cuis.changes $ide.changes
../cogspur/squeak $ide -s CuisApp/src/setupIDE.st
```

Image configuration

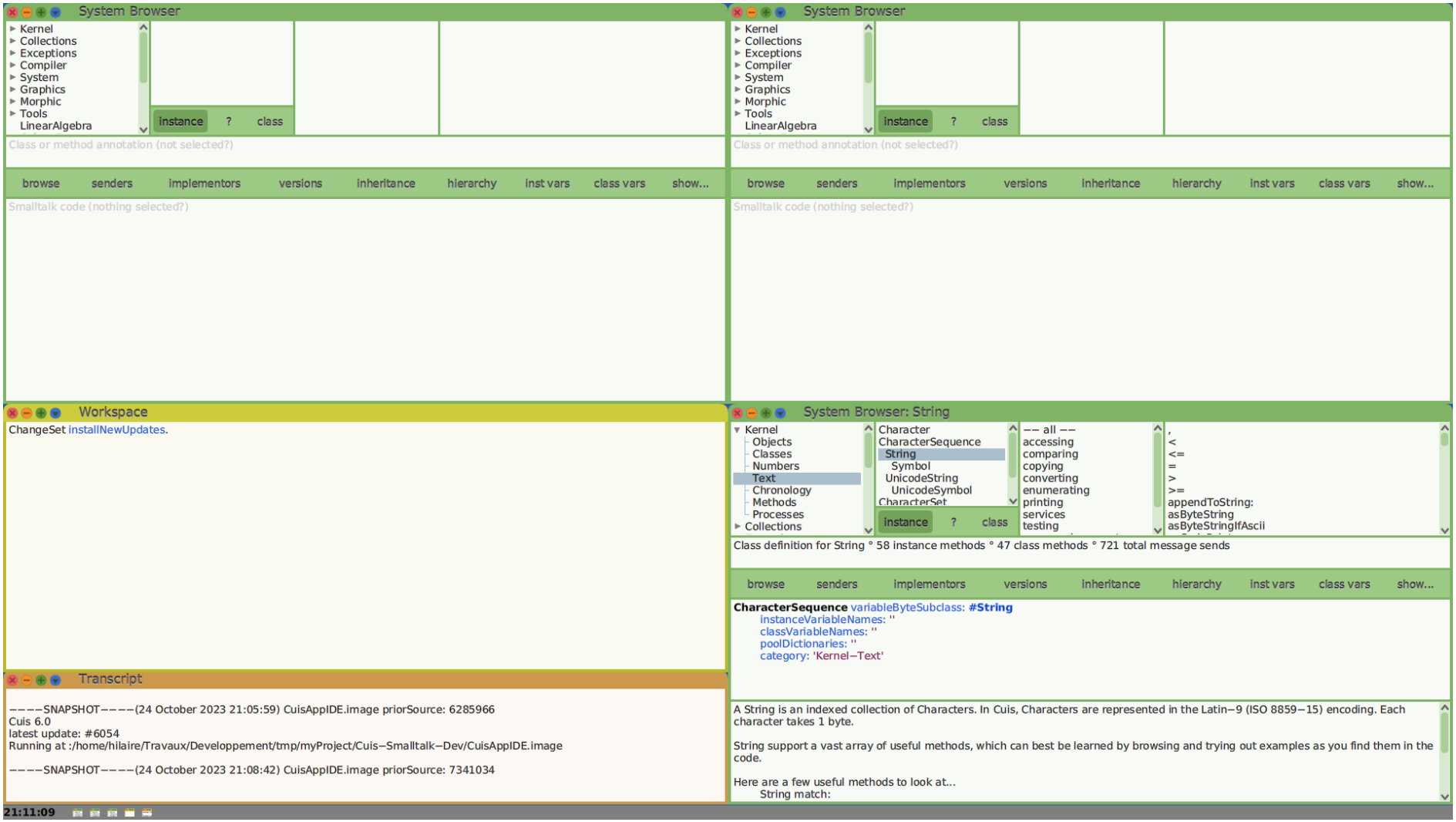
When our `startIDE.sh` Bash script is executed, it starts Cuis with the `setupIDE.st` Smalltalk instructions to install our development environment:

- Start your IDE:

  ```
  cd Cuis-Smalltalk-Dev
  ./CuisApp/startIDE.sh
  ```

- install latest Cuis update
- adjust a few preferences: log, author, font size
- install our CuisApp package
- delete all windows
- open and layout 5 windows: 3 Class Browsers, a Transcript and a Workspace with default content

Installed


System Browser
Kernel
Collections
Exceptions
Compiler
System
Graphics
Morphic
Tools
LinearAlgebra
instance
?
class
Class or method annotation (not selected?)
browse
senders
implementors
versions
inheritance
hierarchy
inst vars
class vars
show...
Smalltalk code (nothing selected?)
Workspace
ChangeSet installNewUpdates.
Transcript
----SNAPSHOT----(24 October 2023 21:05:59) CuisAppIDE.image priorSource: 6285966
Cuis 6.0
latest update: #6054
Running at :/home/hilaire/Travaux/Developpement/tmp/myProject/Cuis-Smalltalk-Dev/CuisAppIDE.image
----SNAPSHOT----(24 October 2023 21:08:42) CuisAppIDE.image priorSource: 7341034
21:11:09
System Browser
Kernel
Collections
Exceptions
Compiler
System
Graphics
Morphic
Tools
LinearAlgebra
instance
?
class
Class or method annotation (not selected?)
browse
senders
implementors
versions
inheritance
hierarchy
inst vars
class vars
show...
Smalltalk code (nothing selected?)
System Browser: String
Kernel
Objects
Classes
Numbers
Text
Chronology
Methods
Processes
Collections
Character
CharacterSequence
String
Symbol
UnicodeString
UnicodeSymbol
CharacterSet
instance
?
class
-- all --
accessing
comparing
copying
converting
enumerating
printing
services
testing
,
<
<=
=
>
>=
appendToString:
asByteString
asByteStringIfAscii
Class definition for String ° 58 instance methods ° 47 class methods ° 721 total message sends
browse
senders
implementors
versions
inheritance
hierarchy
inst vars
class vars
show...
CharacterSequence variableByteSubclass: #String
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Kernel-Text'
A String is an indexed collection of Characters. In Cuis, Characters are represented in the Latin-9 (ISO 8859-15) encoding. Each character takes 1 byte.
String support a vast array of useful methods, which can best be learned by browsing and trying out examples as you find them in the code.
Here are a few useful methods to look at...
String match:

Which ones?

There are best practices, our CuisApp:

- **MVP.** Model-View-Presenter for GUI application.
- **Strategy.** Present a set of methods with differents implementation depending on the platform.
- **Command.** To undo/redo operation for construction, delete, property, merge, move commands

$\Rightarrow$ If you are serious about developing an application, you sould read a book about design patterns[4].

In Dr. Geo

A few more in Dr. Geo:

- **Factories.** It manufactures new models, keeps record of the manufactured models and check for duplicated models.
- **Tools.** Tool and state objects to process multiple-steps user input in the canvas. Tools is instantiated by the presenter, once the user selected a given construction menu.
- **Builder.** Model builders. Fed by the user input via the selected tool to gradually construct an mathematics item.

General considerations

In MVP[3] there are three objects:

- **Model (Domain object).** It represents the domain. It does not know about the view and presenter.
- **View.** It shows the model and the controls (buttons, etc.) to manipulate it. It knows about the presenter.
- **Presenter.** It handles the operationss on the model. It knows about the model and the view.

$\Rightarrow$ The controls in the views are handled by the presenter. Models are unaware of the views and presenter.

$\Rightarrow$ The Presenter is the entry point in the application.

In CuisApp

Our **Domain** object is around a fileEntry:

```
Object subclass: #AppDomain
    instanceVariableNames: 'fileEntry'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CuisApp-Model'
```

Our **View** is around an ImageMorph instance:

```
SystemWindow subclass: #AppView
    instanceVariableNames: 'image presenter'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CuisApp-View'
```

And the **Presenter**, knows about the domain and view through the controlsManager:

```
Object subclass: #App
    instanceVariableNames: 'domain controlManager cmdManager'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CuisApp-Presenter'
```

Presenter is central

Instantiating the Presenter start the application:

```
App>>initialize
    domain := AppDomain new.
    controlManager := AppControlManager for: self.
    controlManager installAppView.
    cmdManager := AppCommandManager new presenter: self.
    self view openInWorld
```

We use an auxiliary object `AppControlManager` to arrange the view: we keep low the complexity in our `AppView` and to implement alternate arrangement by subclassing `AppControlManager`.

Controls

The controls are installed in the view by the AppControlManager:

```
AppControlManager>>installAppView
| scroller |
    view := AppView for: presenter.
    scroller := PluggableScrollPane new :
        layoutSpec: LayoutSpec useAll;
        scroller: view imageMorph.
    view addMorph: self appToolbar layoutSpec: (LayoutSpec
        fixedHeight: AppSystem iconToolbarSize + 4).
    view addMorph: scroller
```

The toolbar

```
AppControlManager>>appToolbar
| toolbar |
    toolbar := LayoutMorph newRow
        separation: 5;
        yourself.
    self appTools do: [:aTool |
        aTool == #spacer
            ifTrue: [toolbar addMorphUseAll: self class spacer]
            ifFalse: [toolbar addMorph: (self button: aTool )]].
↑ toolbar
```

Each tool is described in its own method:

```
AppControlManager>>openButtonData
" label - iconName - callback - description "
↑ {'Open' translated . #open . #openImage .
   'Select a picture to visualize and to operate with.' translate
```

Observe the `#translated` message sent to the tool label and description.

The tool

Observe, the model of the button is the Presenter, action is sent to it:

```
AppControlManager>>button: symbol
"array first = menu label or button label
array second = button form = section mode
array third = symbol callback
array fourth = help string"
| array |
    array := self perform: (symbol, #ButtonData) asSymbol.
    ↑ ButtonMorph
    model: presenter
        action: array third ::
        enableSelector: (
            symbol == #open ifTrue: [true] ifFalse: [#isButtonActive]);
        icon: (icons get: array second size: AppSystem iconToolbarSize);
        setBalloonText: array fourth;
        ../..
```

Callback

Let's take a look to the #openImage callback:

```
App>openImage
| answer |
    answer := (StandardFileMenu new
        oldFileFrom: AppSystem picturesPath
        withPattern: '*.png *.jpg *.jpeg'
        excludePattern: '.*')
            startUpWithCaption: 'Pick up a picture' translated.
  answer ifNotNil: [
      domain fileEntry: answer directory // answer name.
      cmdManager release]
```

$\Rightarrow$ The domain is updated with the fileEntry of the newly selected picture.

Do you think there is a missing piece?

Decoupling Domain & View

How the View knows about a newly selected image? We use the Observer pattern. The Domain triggers a #newImageSelected event:

```
AppDomain>>fileEntry: aFileEntry
    fileEntry := aFileEntry.
    self triggerEvent: #newImageSelected
```

⇒ Our View is listening to the #newImageSelected event:

```
AppView>>initialize
../..
    self domain when: #newImageSelected
      send: #loadImage to: self
```

⇒ And the method loadImage is called:

```
AppView>>loadImage
    image image: (ImageReadWriter
        formFromFileEntry: self domain fileEntry).
    self update: #relabel
```

General considerations

Three objects:

- A command manager, `AppCommandManager`, to execute specific command
- A command stack, `AppCommandStack`, where are stacked the command instances
- A hierarchy of command, `AppCommand`, each specialized in one type of operation

Two important command methods: `execute` and `unexecute`
Let's observe it with the rotate right operation.

Rotate right callback

When the button is pressed in the toolbar, the Presenter's method is executed:

```
App>>rotateRight
    cmdManager rotateRight
```

The command manager create the command, add it to the stack then execute it:

```
AppCommandManager>>rotateRight
|command|
    command := stack nextPut:
        (AppRotateCommand presenter: presenter).
    command degrees: 90.
    ↑ command execute
```

Rotate right command

Observe the execute method. As the execute command is not, the unexecute method just rotate the image in the other direction:

```
AppRotateCommand>>execute
    self imageMorph image: (self rotatedBy: degrees)

AppRotateCommand>>unexecute
    self imageMorph image: (self rotatedBy: degrees negated)
```

⇒ For destructive command as zoom out – we definitely lose image pixels – we have to keep a copy of the original image:

```
AppZoomOutCommand>>execute
    cacheForm := self imageMorph form.
    self imageMorph image: (cacheForm magnifyBy: 0.5)

AppZoomOutCommand>>unexecute
    self imageMorph image: cacheForm
```

Modus operandi

Your application in native language. It need the “Gettext” and “System-Locales” Cuis packages.

- In the code, you write string in English.
- You send the message `#translated` to each string you want to be translated:

```
... startUpWithCaption: 'Pick up a picture' translated.
```

- You define your translation domain based on Class category:

```
TextDomainManager
    registerCategoryPrefix: 'CuisApp'
    domain: 'cuisapp'.
```

- You export a .pot template file:

```
GetTextExporter exportTemplate
```

It creates a file Cuis-Smalltalk-Dev/po/cuisapp/cuisapp.pot

- You copy it as “es.po” and translate it to your native language


CuisApp template


Dr. Geo repository at launchpad with translation management


https://www.martinfowler.com/eaaDev/uiArchs.html


Sherman R. Alpert, Kyle Brown, Bobby Woolf, The Design Patterns Smalltalk Companion, Addison-Wesley, 1998


Setting up and starting Cuis Smalltalk