Design GUI with Morph - (DRAFT)

A Practical Guide for Cuis

The screen is a window through which one sees a virtual world. The challenge is to make that world look real, act real, sound real, feel real.
—*Ivan Sutherland*

H. Fernandes

“Design GUI with Morph - (DRAFT)” booklet is for Cuis-Smalltalk (7.0 or later), a free and modern implementation of the Smalltalk language and environment.

Copyright © 2025 H. Fernandes
Thanks to Mark Volkmann to share his examples and chapter on layout.

Thanks to Ken Dickey, Thiago da Silva Lino for the reviews of the booklet, suggestions and borrowed texts. Your help is very valuable.

Compilation : 5 August 2025

Documentation source: `https://github.com/DrCuis/DesignGUI`

Table of Contents

1 Introduction

This booklet is a collection of how-to guides to learn how to build basic GUI (Graphic User Interface) with Cuis-Smalltalk and its Morphic system. Some chapters, such as Chapter 3 [What is a Morph after all?], page 11, and Chapter 5 [Design a GUI application], page 43, present deeper knowledge as well. Most chapters can be read independently to serve the needs of the reader.

The standard Cuis-Smalltalk framework for building user interfaces is Morphic 3; it is the most refined version of the Morphic graphic system found in Smalltalk environments. The classes provided by this framework are in the class category `Morphic`. What non-Smalltalk GUI libraries refer to as *components* or *widgets* are called *Morphs* in Cuis-Smalltalk; therefore, in the rest of this book, the terms Morph, widget, and component are interchangeable.

1.1 Overview

There are many provided morphs that are subclasses of the class Morph, and it's easy to create custom morphs.

The primary way of organizing morphs is to create instances of LayoutMorph and add submorphs to them. Instances of LayoutMorph can be nested to achieve pretty much any layout. It is used massively; in a Cuis-Smalltalk image with several browsers open, there are tens of layouts.

Execute the code below with ***Ctrl-p*** to get the result printed in your image:

```
LayoutMorph allInstances size
⇒ 79.
```

Open a Workspace and execute the following expressions:

```
box := ColoredBoxMorph new.
box color: Color pink.
box morphExtent: 200 @ 100.
box openInWorld
```

Example 1.1: My first morph

to render a pink rectangle.

Open the halo of work-handles for this morph by holding down a key and clicking it. The key to hold down depends on your platform. It is the ***Command*** key in macOS and the ***Control*** key or ***Shift+Control*** key in other platforms. With a three-button mouse, you can middle-click to get the halo. Click the red button in the upper-left of the halo to remove the morph.

1.2 Quick Start

Rendering a single morph isn't particularly interesting. Let's render some labels, a text input, and a button to implement a simple application.

In a new Workspace, select and invoke the following code. Use ***Ctrl+a*** to select All, then ***Ctrl+d*** to DoIt.

```
| view input greetingLabel button row |

view := LayoutMorph newColumn.

"Create a single-line text input.
 Using zero for the height causes it to calculate
 the minimum required height for a single line."
input := TextModelMorph withText: '' ::
    hideScrollBarsIndefinitely;
    morphExtent: 200 @ 0;
    wrapFlag: false.

greetingLabel := LabelMorph contents: '' ::
    color: Color red;
    font: FontFamily defaultFamilyAndPointSize bold.
"Create a button that when clicked executes
the associated block of code"
button := PluggableButtonMorph
    model: [greetingLabel contents: 'Hello, ', input text, '!']
    action: #value.
button label: 'Greet'.

row := LayoutMorph newRow
    gap: 10;
    addMorph: (LabelMorph contents: 'Name:');
    addMorph: input;
    addMorph: button.

view
    padding: 20;
    addMorph: row;
    addMorph: greetingLabel;
    openInWorld
```

Example 1.2: My first layout

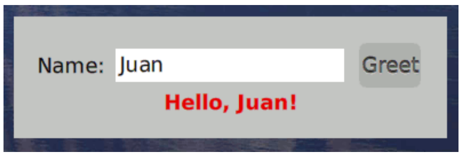


Figure 1.1: My first interface

This morph is a bit like a “Hello World” panel; it should be inserted in a window with a proper title and icon to operate on. To remove this panel, you need to again request its halo, which is not very practical. Let’s change the simple app we’ve built so it can be inside a window. In the previous code, edit the end of the script to insert our view in a SystemWindow instance.

Again, select the entire script and invoke it.

```
[...]
view
    padding: 20;
    addMorph: row;
    addMorph: greetingLabel.

SystemWindow new ::
   setLabel: 'Hello World!';
   addMorph: view;
   openInWorld;
   morphExtent: 300@100
```

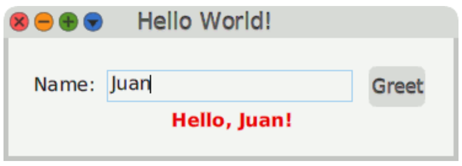


Figure 1.2: My first window

2 Layout Components

How to arrange components together is a cornerstone of GUI design. A text entry may come with a text label to indicate what input is expected from the user. These are two widgets – one passive text label and one active text input entry – the GUI designer needs to decide how to arrange them. In a column? In a row? Centered? This chapter covers this topic.

Instances of the class LayoutMorph actively manage the position and size of morphs that are added to them, referred to as submorphs.

2.1 To Row or Not to Row

LayoutMorph instances arrange submorphs in a row or column and are created with LayoutMorph newRow or LayoutMorph newColumn. An instance cannot be created with LayoutMorph new because a direction must first be decided upon.

The submorphs must be instance subclasses of PlacedMorph. The LayoutMorph is such a class, so these can be nested, which enables achieving practically any layout.

Let's layout three instances of ColoredBoxMorph. Using this kind of morph for examples allows us to clearly see their bounds:

```
box1 := ColoredBoxMorph new color: Color red; morphExtent: 50 @ 50.
box2 := ColoredBoxMorph new color: Color green; morphExtent: 75 @ 75.
box3 := ColoredBoxMorph new color: Color blue; morphExtent: 100 @ 100.
layout := LayoutMorph newRow
    addMorph: box1;
    addMorph: box2;
    addMorph: box3;
    openInWorld
```

Example 2.1: Layout base example

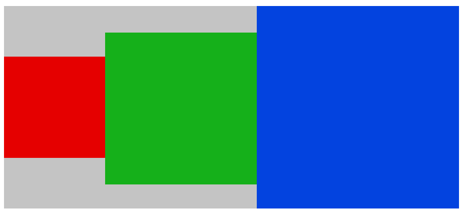


Figure 2.1: Basic use of LayoutMorph

By default, there is no space between the edges of the LayoutMorph and its submorphs, and there is no space between the submorphs.

To add 10 pixels of space between the edges of the LayoutMorph and its submorphs, send layout padding: 10. If the argument is a Point, its x value is used for left and right padding, and its y value is used for top and bottom padding.

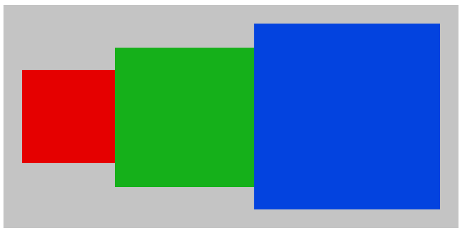


Figure 2.2: Adding padding

To add 10 pixels of space between the submorphs, send layout gap: 10.

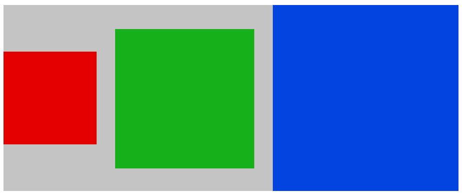


Figure 2.3: Adding gap

To add both padding and gap in a single message, do layout separation: 10.

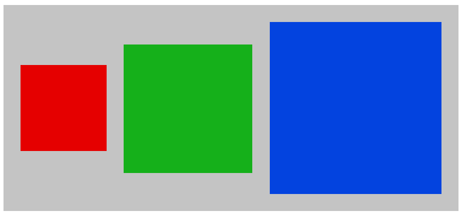


Figure 2.4: Separation of 10 is both gap and padding of 10

2.2 Alignment

For a row LayoutMorph, the major axis is x and the minor axis is y. For a column LayoutMorph, the major axis is y and the minor axis is x.

If a LayoutMorph is given a width or height that is larger than needed to fit the submorphs, we can specify how the submorphs should be aligned. By default, the submorphs will be aligned at the beginning of its major axis and centered on its minor axis.

You can lengthen the layout by adding space to its morphExtent: layout morphExtent: (layout morphExtent + (80 @ 0)). Another way of doing this is to get the LayoutMorph's halo and drag its yellow Change size button at the lower right.

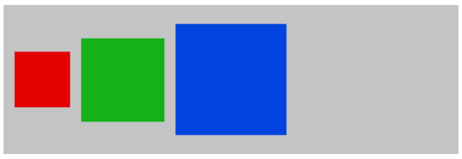


Figure 2.5: Default alignment

To change the major axis alignment, **send to the layout morph** the message #axisEdgeWeight: with a floating point argument between 0 and 1. The argument should be 0 for left-align, 0.5 for center, or 1 for right-align. The following symbols can also be used for the argument:

- **For rows.** #rowLeft, #center, or #rowRight,
- **For columns.** #columnTop, #center, or #columnBottom.

```
layout axisEdgeWeight: #center.
```

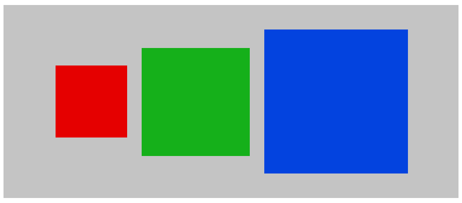


Figure 2.6: Center alignment

To change the minor axis alignment, send to each submorph the message #offAxisEdgeWeight: with a floating point argument. It takes the same argument values as the #axisEdgeWeight: message. Again, this message must be sent to each of the submorphs, not to the LayoutMorph:

```
    layout submorphsDo: [:submorph |
      submorph layoutSpec offAxisEdgeWeight: 0]
```

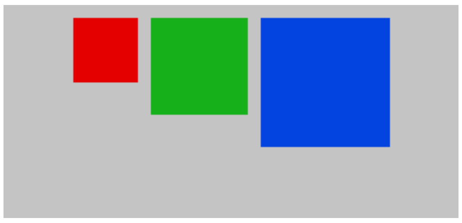


Figure 2.7: Top alignment

Alternatively, to numeric value arguments, symbols can be used:

- **For rows.** #rowTop, #center, #rowBottom,
- **For columns.** #columnLeft, #center, #columnRight.

Browse the method offAxisEdgeWeight: to discover more symbols.

Edit the example Example 2.1 so that box1 sits at the top of its owner, box2 at the middle, and box3 at the bottom.

Exercise 2.1: Top to down

2.3 Proportion

In all the examples so far, the submorphs each have a fixed size. They can also be given proportional sizes so their actual size will be based on a percentage of the available space. The available space is the space after removing the padding, the gaps between submorphs, and the sizes of submorphs that have a fixed size.

Each submorph can be given a different proportional size value. The amount of the available space given to each is its percentage of the total values. For example, suppose a row LayoutMorph contains three submorphs that use a proportional width, and they are assigned the values 2, 3, and 5. The total is 10, so the first will take 2/10 (20%), the second will take 3/10 (30%), and the third will take 5/10 (50%) of the available space.

Let's modify the previous example to cause the second submorph to take all the available space.

Each morph can have a LayoutSpec that specifies how it should be laid out within its owner; this is ignored if the owner is not a LayoutMorph instance. LayoutSpec is abstract with two subclasses:

- LayoutSizeSpec - for rows/columns. Manages dimensions via fixedWidth, fixedHeight, proportionalWidth and proportionalHeight and alignment via offAxisEdgeWeight.
- LayoutEdgesSpec - for form layouts. Defines corner anchors using topLeftEdgesWeight, bottomRightEdgesWeight, topLeftOffset and bottomRightOffset.

Instances of LayoutSpec have morph as instance variable. There are methods to set each of these instance variables.

```
box2 layoutSpec proportionalWidth: 1
```

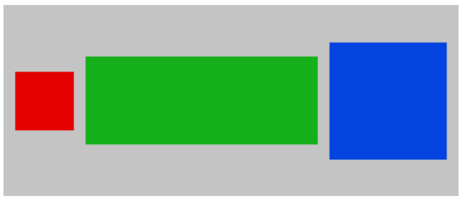


Figure 2.8: Using proportionalWidth

We could have added box2 to layout using the message #add:proportionWidth: which adds a submorph AND sets the proportionalWidth property of its LayoutSizeSpec. However, an issue with this approach is that it creates a new instance of LayoutSizeSpec where the values of proportionalWidth and proportionalHeight are both 1. To only modify one of those properties, it's best to set it directly on the LayoutSizedSpec instance of the morph.

Let's modify our example Example 2.1 so the boxes are spread across the width of the LayoutMorph instance with an even amount of space between them.

```
box1 := ColoredBoxMorph new color: Color red; morphExtent: 50 @ 75.
box2 := ColoredBoxMorph new color: Color green; morphExtent: 75 @ 50.
box3 := ColoredBoxMorph new color: Color blue; morphExtent: 100 @ 100.
spacer1 := ColoredBoxMorph new color: Color transparent.
spacer1 layoutSpec proportionalWidth: 1.
spacer2 := ColoredBoxMorph new color: Color transparent.
spacer2 layoutSpec proportionalWidth: 1.
layout := LayoutMorph newRow
   morphExtent: 350 @ 150;
   separation: 10;
   addMorph: box1;
   addMorph: spacer1;
   addMorph: box2;
   addMorph: spacer2;
   addMorph: box3.
layout openInWorld.
```

Example 2.2: Layout with spacer

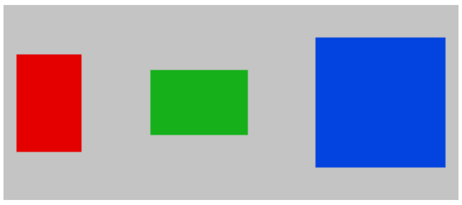


Figure 2.9: Evenly spaced

spacer1 and spacer2 each use 1 of the free space left by the other morphs in the owner. The total free space is counted as 2 (1+1), therefore each spacer will occupy 1/2 of the free space.

Edit the example Example 2.2 so that the first spacer uses one quarter of the free space and the second one three quarters.

Exercise 2.2: Spacers in quarters

2.4 Nest of Layouts

To wrap up our discussion on using the LayoutMorph class, let's look at an example that nests layouts:

```
column1 := LayoutMorph newColumn
    addMorph: (LabelMorph contents: 'Apple');
    addMorph: (LabelMorph contents: 'Banana');
    addMorph: (LabelMorph contents: 'Cherry').
column1 layoutSpec proportionalHeight: 0. "defaults to 1"

column2 := LayoutMorph newColumn
    addMorph: (LabelMorph contents: 'Spring');
    addMorph: (LabelMorph contents: 'Winter');
    addMorph: (LabelMorph contents: 'Summer');
    addMorph: (LabelMorph contents: 'Fall').
column2 layoutSpec proportionalHeight: 0. "defaults to 1"

row := LayoutMorph newRow
    separation: 20;
    addMorph: column1;
    addMorph: (LabelMorph contents: 'What are your favorites?');
    addMorph: column2.

row openInWorld.
```

Example 2.3: Nesting layouts

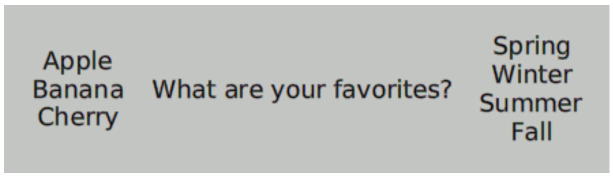


Figure 2.10: Nested LayoutMorphs

We can add a colored border to any morph that is an instance of a subclass of BorderedBoxMorph. This is useful for debugging. Let's add borders to some of the morphs in the previous example Example 2.3:

```
column1 := LayoutMorph newColumn
   addMorph: (LabelMorph contents: 'Apple');
   addMorph: (LabelMorph contents: 'Banana');
   addMorph: (LabelMorph contents: 'Cherry');
   borderColor: Color red; borderWidth: 2.
column1 layoutSpec proportionalHeight: 0.

column2 := LayoutMorph newColumn
   addMorph: (LabelMorph contents: 'Spring');
   addMorph: (LabelMorph contents: 'Winter');
   addMorph: (LabelMorph contents: 'Summer');
   addMorph: (LabelMorph contents: 'Fall');
   borderColor: Color blue; borderWidth: 2.
column2 layoutSpec proportionalHeight: 0.

center := LabelMorph contents: 'What are your favorites?' ::
   borderColor: Color green; borderWidth: 2.

row := LayoutMorph newRow
   separation: 20;
   addMorph: column1;
   addMorph: center;
   addMorph: column2.

row openInWorld.
```

Example 2.4: Revealed nested layouts

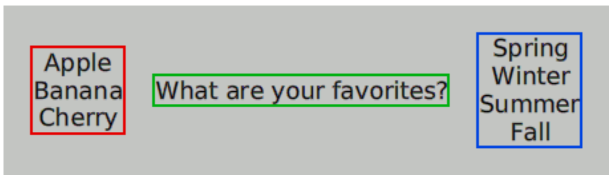


Figure 2.11: Morph borders

3 What is a Morph after all?

The Cuis' Morph framework is fundamental to Cuis-Smalltalk. Every single visual piece the user sees and interacts with in the Cuis' window is a morph. A morph offers both visual representation and interactivity with keyboard and mouse inputs.

3.1 A World of Morphs

A bit of introspection reveals how many Morph instances are in use in the Cuis-Smalltalk living system

```
PluggableButtonMorph allInstances size
⇒ 288
```
[1]

Example 3.1: How many buttons?

The previous example asked for one specific type of morph. What about asking for all types of morph?

```
Morph allSubclasses inject: 0 into: [:count :aClass |
  count + aClass allInstances size]
⇒ 1558
```

Example 3.2: How many morphs are operating on the system?

This number, likely different on your own Cuis-Smalltalk system, represents all the objects necessary to operate visually the system. This count changes all the time; new morphs are created, and ones no longer necessary are garbage collected regularly. You can play a bit with the system and evaluate again this count.

3.1.1 Tree of Morphs

Any morph can contain other morphs. We already know a LayoutMorph contains several morphs and manage the way there are layed out; these is reflected in the submorphs attribute of each morph, when this collection is empty it means there is no sub-morph. In the other hand, any morph being displayed knows about its containing owner: the owner attribute of each morph refers to the morph owning it. And as you can expect it, the owner morph submorphs attribute contains the owned morph as well.

Then what happens when the owner attribute is nil? The morph is simply not visible! This double link between owner and owned is very convenient and necessary for the morph framework depending on the situation.

For example, while an owned morph may have its own local coordinates system for its drawing operations, it can refer to its owner to know about its global coordinates situation, from the point of view of the owner or even from the World perspective – *simplified for readiness*

[1] As a recall, you *execute-and-print* to get the result printed; otherwise, it is only executed. Use shortcut `Ctrl-p` (PrintIt) instead of `Ctrl-d` (DoIt).

```
Morph>>externalizeToWorld: aPoint
    "aPoint is in own coordinates. Answer is in world coordinates."
    | inOwners |
    inOwners := self externalize: aPoint.
    ↑ owner externalizeToWorld: inOwners
```

Access to the owner is also useful for mundane aspects like style. A morph asks its owner's color to draw itself accordingly:

```
MenuLineMorph>>drawOn: aCanvas
    | baseColor |
    baseColor := owner color.
    aCanvas
       fillRectangle: (`0@0` corner: extent x @ (extent y / 2))
       color: baseColor twiceDarker.
    ...
```

On the other hand, the owner, depending on its nature, can decide how to dispose of its sub-morphs, like a LayoutMorph does by first requesting which sub-morphs to lay out:

```
LayoutMorph>>submorphsToLayout
    "Select those that will be laid out."
    ↑ submorphs select: [ :m | m visible ]
```

3.1.2 Halo of Icons

Because everything is an object in Cuis-Smalltalk, every visual part of the system is represented through an object the user can interact with and inspect. The Halo system, designed with objects, is a special visual tool to know more about specific Morph instances. It is invoked on any Morph instance by a middle button click; a halo of colored icons then shows up, surrounding the most general selected morph.

Succeeding clicks access more interior submorphs under the mouse pointer, and the halo is then updated accordingly. When pressing the *Shift* key, the direction of morph selection is changed: the owner of the selected morph is then selected instead.

Each icon gives access to specific actions to operate on the morph.

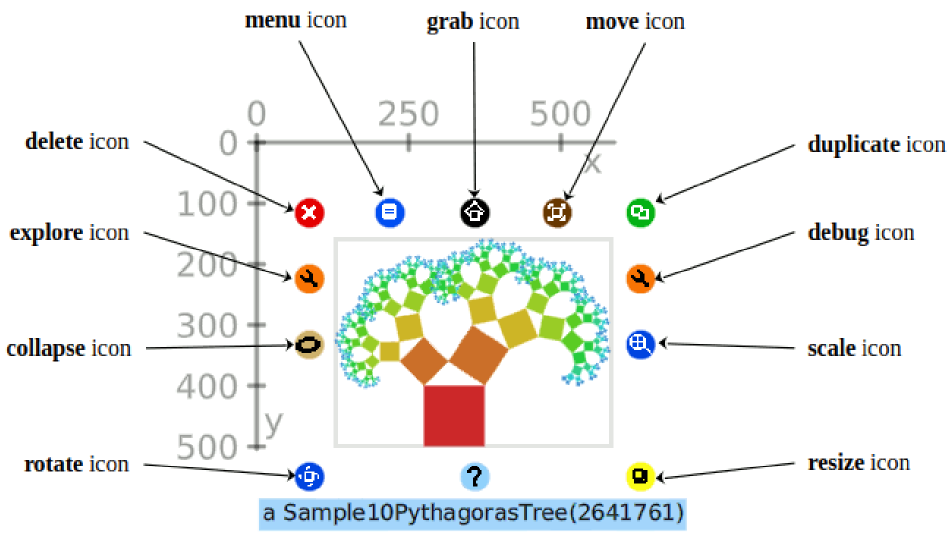


Figure 3.1: Halo and descriptions on each icon functions

Some actions need clarifications:

- **grab.** It grabs the morph from its owner and can be dropped in the World or any other morph accepting dropped morphs.
- **move.** It moves the morph within its owner. Try the differences between grab and move with a menu entry in the World menu.
- **resize.** On a text morph, it changes the area of the morph; on more graphics-oriented morphs like the Clock example in Cuis-Smalltalk, it scales the morph.
- **explore.** It opens directly the explorer tool discussed earlier.

We previously noted the submorph and owner relationship among morphs. In a halo, the orange wrench icon at the right gives access to a set of actions to explore these relations; try it!

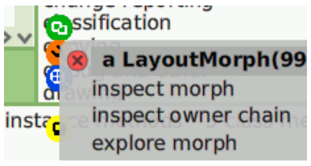


Figure 3.2: Actions to explore sub-morph and owner relations

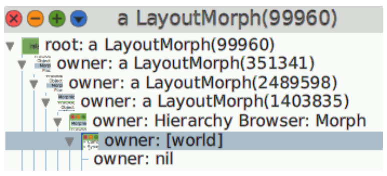
3.1.3 The Special World Morph

In Cuis-Smalltalk, there is a special morph, a WorldMorph instance, representing the top morph of Cuis-Smalltalk:

```
WorldMorph allInstances
⇒ { [world]}
```

Example 3.3: There is only one World!

Guess what? It is the only morph without an owner and still visible. Take any morph in the world, invoke its halo, then from the wrench icon, select the menu entry `explore morph` and browse the chain of owners until you reach the World; you discover it does not have an owner:



```
a LayoutMorph(99960)
▼ root: a LayoutMorph(99960)
  ▼ owner: a LayoutMorph(351341)
    ▼ owner: a LayoutMorph(2489598)
      ▼ owner: a LayoutMorph(1403835)
        ▼ owner: Hierarchy Browser: Morph
          ▼ owner: [world]
              owner: nil
```

Figure 3.3: The chain of owners of a morph

Execute the following code to experiment on a morph from its inspector:

```
Morph new ::
  inspect;
  openInWorld
```

Example 3.4: Create a morph, inspect it and open it in the World

A blue rectangular morph shows up at the top left corner of the Cuis-Smalltalk window. Sometime, the morph we are playing with is out of sight, or we are not sure where it is; a convenient way to find is to ask it flashes itself. In the inspector, execute the code `self flash`.

Any morph knows about its World, in the inspector execute: `self runningWorld`. As a World is a kind of Morph, we can flash it too: `self runningWorld flash`.

3.2 Morph Hierarchy

The Morph class is the root class to all kinds of Morph, sub-classes specialize on specific facets or add some features. Let's learn a bit more about the Morph class itself and then explore its most fundamental sub-classes.

In a stock Cuis-Smalltalk system, browse the Morph class; it is in the `Morphic>Kernel` class category. Here are the most fundamental types of Morphs:

- Morph
    - PlacedMorph
        - BoxMorph
            - ColoredBoxMorph
                - BorderedBoxMorph
                - PasteUpMorph

- WorldMorph

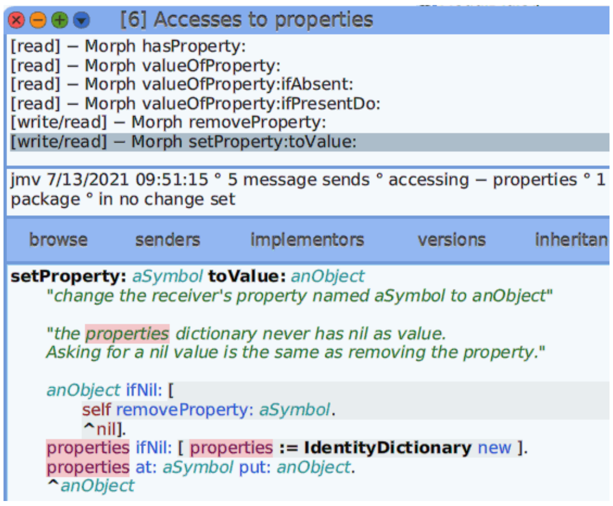
- HandMorph

Observe how the Morph has only PlacedMorph as its subclass; this is odd and requires some scrutiny by looking at the differences between these two classes. Indeed, if Morph has only one subclass, the two should be merged.

Let's examine the attributes of the Morph class:

```
Object subclass: #Morph
   instanceVariableNames: 'owner submorphs properties id privateDisplayBounds'
   classVariableNames: 'LastMorphId'
   poolDictionaries: ''
   category: 'Morphic-Kernel'
```

There are the expected owner and submorphs attributes; the properties attribute is set, when needed, as a dictionary to dynamically add attributes or behaviors – with appropriate bloc of code – to any Morph instances. There are companion methods to set or retrieve a property.



```
[read] – Morph hasProperty:
[read] – Morph valueOfProperty:
[read] – Morph valueOfProperty:ifAbsent:
[read] – Morph valueOfProperty:ifPresentDo:
[write/read] – Morph removeProperty:
[write/read] – Morph setProperty:toValue:
```

jmv 7/13/2021 09:51:15 ° 5 message sends ° accessing – properties ° 1 package ° in no change set

```
setProperty: aSymbol toValue: anObject
	"change the receiver's property named aSymbol to anObject"

	"the properties dictionary never has nil as value.
	Asking for a nil value is the same as removing the property."

	anObject ifNil: [
		self removeProperty: aSymbol.
		^nil].
	properties ifNil: [ properties := IdentityDictionary new ].
	properties at: aSymbol put: anObject.
	^anObject
```

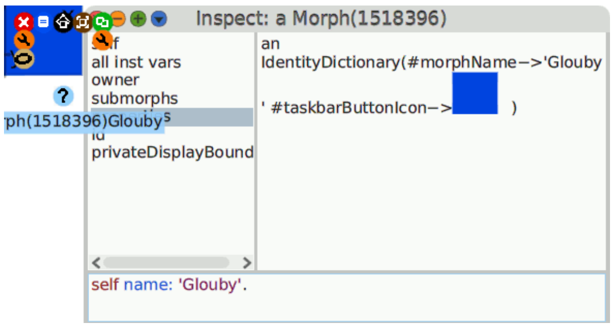
Figure 3.4: Morph's methods to manipulate properties

This properties attribute is used a lot. For example, observe its use case to set and get a specific name to a Morph instance

```
name: anObject
   "Set the morphName property"
   self setProperty: #morphName toValue: 'Glouby'
```

```
name
   "Answer the value of morphName"
   ↑ self valueOfProperty: #morphName
```

Create a new morph as seen in Example 3.4, then in the inspector execute self name: 'Glouby'. Observe how properties was amended with the dictionary entry #morphName->'Glouby'; when invoking the morph's halo, its new name is printed too.



Inspect: a Morph(1518396)

```
self                     an
all inst vars            IdentityDictionary(#morphName->'Glouby
owner
submorphs                ' #taskbarButtonIcon->        )
properties
id
privateDisplayBound
```

self name: 'Glouby'.

Figure 3.5: A morph's properties

Back to the Morph definition, you can observe there is no attribute to describe the shape of a morph![2] Indeed, a morph has no pre-established shape; its shape is described dynamically – at execution time – through its drawOn: method:

```
drawOn: aCanvas
   drawOn: aCanvas
      aCanvas
         fillRectangle: '-75 @ -70 corner: 75 @ 70'
         color: 'Color blue'
```

A Morph instance's drawing instructions are relative to its owner's coordinates system. If you observe this blue rectangular morph we created, it is located at the top left corner of the World, where the (0,0) coordinates are; in fact, a large part of this morph is not visible, three quarters of its area is outside of the screen.

If you observe the halo as seen in Figure 3.5, there is no **scale**, **resize**, and **rotate** icons! Indeed, a Morph instance can't be resized or rotated. Moreover, when you try to move the morph through its **move** icon, you can't; it sticks to its position, hard-coded in its drawOn: method. It is about time to introduce the PlacedMorph class, which offers these features.

3.2.1 Morph You Can Move

...but not only.

These abilities of PlacedMorph to scale, to resize, to rotate, and to be moved are some of the differences with Morph. It is about time to examine its definition:

[2] The other attributes are used for internal management of the morphs and you don't need to care about them.

```
Morph subclass: #PlacedMorph
  instanceVariableNames: 'location layoutSpec'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Morphic-Kernel'
```

We have two more instance variables:

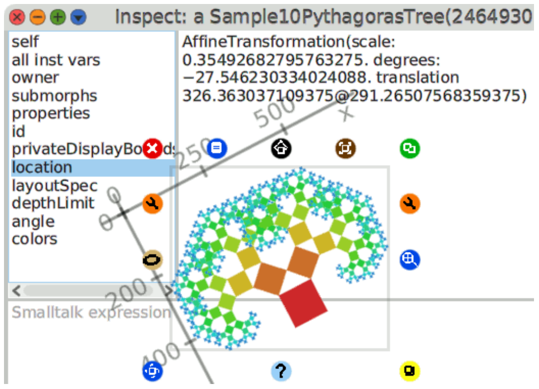
- This layoutSpec attribute is obviously related to the optional layout specification used when a PlacedMorph instance is set in a layout morph; this also tells us a Morph instance can't be set in a layout.
- This location attribute holds the information regarding position, scale, and rotation; it is a kind of affine transformation coded as a matrix[3]. These transformations are applied – in one matrix transformation – relatively to its owner.

Time to experiment a bit! Execute this simple code:

```
Sample10PythagorasTree new ::
  openInWorld ;
  inspect
```

Example 3.5: Inspect a star

to explore with the inspector. Then invoke its halo to move, resize, and rotate to observe how its location attribute is updated:



Inspect: a Sample10PythagorasTree(2464930

```
self
all inst vars
owner
submorphs
properties
id
privateDisplayBounds
location
layoutSpec
depthLimit
angle
colors
```

```
AffineTransformation(scale:
0.35492682795763275. degrees:
−27.546230334024088. translation
326.363037109375@291.26507568359375)
```

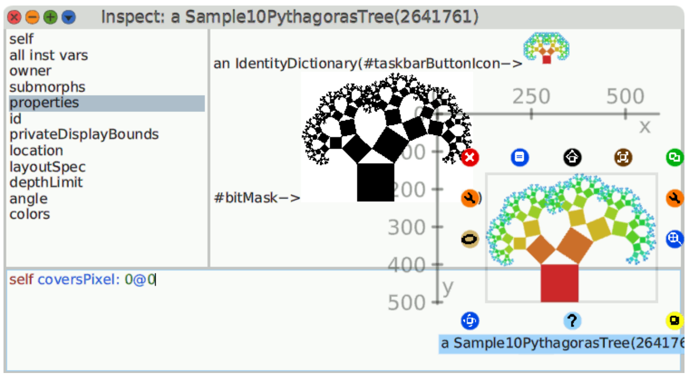
Figure 3.6: location update according to scale, position and rotation

In one attribute, we have the information on position, size, and rotation. Moreover, when sending messages to a morph like #morphPosition, #morphPosition:, #scale:, #rotation:, etc., Cuis-Smalltalk is just querying or adjusting the location matrix parameters. Therefore, you, as the user of a PlacedMorph instance, only use these messages without the need to care about the underlying matrix representation.

[3] `https://en.wikipedia.org/wiki/Affine_transformation#Augmented_matrix`

One last bit of experiment to foster Morph understanding: Cuis-Smalltalk knows about morph detection. In our previous morph's inspector, execute and print self coversPixel: 0@0; the answer is likely false if the morph is not positioned to cover the World origin. Now inspect its properties; there is a bitMask entry referring to a 1-bit depth Form. This is how Cuis-Smalltalk does pixel detection. In this Form, a kind of picture, each bit represents the pixel obstruction of the morph: 0 means nothing painted, 1 means the morph paints the pixel with an arbitrary color we don't need to know about here.

This property is automatically garbage collected when not needed anymore or irrelevant.


Inspect: a Sample10PythagorasTree(2641761)

self
all inst vars
owner
submorphs
properties
id
privateDisplayBounds
location
layoutSpec
depthLimit
angle
colors

an IdentityDictionary(#taskbarButtonIcon->

#bitMask->

self coversPixel: 0@0

Figure 3.7: Pixel detection illustrated

The reader may wonder if there is any use case of the Morph class when its instances can't be grabbed and moved around. Why not just use PlacedMorph? In DrGeo software[4], such morphs are used to represent geometric objects drawn in the coordinates system of the owner morph; their positions and aspects are completely dependent on the underlying mathematical models.

3.2.2 Rectangular Morph

The remaining classes of the kernel morph hierarchy add geometry and visual properties. BoxMorph is the most important one; it is bounded to a rectangular extent. It is the root of most sub-morphs used to construct GUI (Graphic User Interface). Indeed, its elements are most of the time bounded to a rectangular shape. It allows a lot of optimization in the rendering of the whole Cuis-Smalltalk GUI.

```
PlacedMorph subclass: #BoxMorph
  instanceVariableNames: 'extent'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Morphic-Kernel'
```

This additional extent attribute defines its width (x-axis) and height (y-axis) in the morph local coordinates.

There are companion methods to play with:

[4] http://gnu.org/s/dr-geo

```
BoxMorph new ::
  openInWorld;
  morphWidth: 1000;
  morphHeight: 10
```

Example 3.6: Adjust width and height

Or more directly:

```
BoxMorph new ::
  openInWorld;
  morphExtent: 1000@10
```

Example 3.7: Adjust extent!

As this is a kind of PlacedMorph, we ask directly about its bounds – position and extent – by executing and printing the result

```
BoxMorph new ::
  openInWorld;
  localBounds
⇒ 0@0 corner: 50@40.
```

Example 3.8: Is my place occupying a lot of space?

The returned object is a Rectangle instance. As Point instances, this is a fundamental architecture class when dealing with GUI.

Observe as you get the same local bounds even when positioning around the morph:

```
BoxMorph new ::
  openInWorld;
  morphPosition: 200@200;
  localBounds
⇒ 0@0 corner: 50@40.
```

Example 3.9: My bounds do not depend on my position

This is because we are asking in the local morph coordinates system. What the reader may want is asking in the owner coordinates system, here the Cuis-Smalltalk World

```
BoxMorph new ::
 openInWorld;
 fullBoundsInOwner
⇒ 493@594 corner: 543@634
```

Example 3.10: Morph positioned at pointer position

```
BoxMorph new ::
   openInWorld;
   morphPosition: 200@200;
   fullBoundsInOwner
⇒ 200@200 corner: 250@240.
```

Example 3.11: Morph positioned according to programming instruction

There are several methods dealing specifically with BoxMorph; take a look at them and play a bit to foster your understanding. A last bit about this class: earlier at Figure 3.7, we discussed pixel detection with a complicated morph containing holes. In this situation, the detection is done with an expensive computed mask. BoxMorph optimizes this thanks to its rectangular shape:

```
coversPixel: worldPoint
   "Answer true as long as worldPoint is inside our shape even if:
      - a submorph (above us) also covers it
      - a sibling that is above us or one of their submorphs also covers it.
   This implementation is cheap, we are a rectangular shape."
   ↑ self coversLocalPoint: (self internalizeFromWorld: worldPoint)
```

By browsing the implementors of this method, we can read how different it is for a non-rectangular morph.

The other morph classes in the kernel category, ColoredBoxMorph and BorderBoxMorph, add cosmetic features like color, border width and color, padding to set space between the border and inner content. It doesn't require much explanation. PasteUpMorph and WorldMorph are rarely used directly.

This modest dive into the Morph framework is now over; we hope it provided enough background to foster your understanding when building GUI. In another booklet, we will discuss more deeply the Morph framework.

4 Handle User Interaction

This chapter covers the use of widgets in the base Cuis-Smalltalk image. We hope the examples presented in this chapter provide enough knowledge to further explore the Cuis-Smalltalk widgets and tools present in the base image. We discuss core functionalities such as associating a widget to a model, handling user interactions, and updating a widget from a model. We conclude the chapter with the fully functional Appendix F [Memory Game v1], page 90.

4.1 Button

Perhaps the simplest user interaction to implement is responding to button clicks. The PluggableButtonMorph supports this.

Several of the Morph subclasses are "pluggable". This means they can be configured through composition rather than inheritance. Configuration is achieved by specifying a model object and a selector for messages to be sent to the model object when an interaction occurs.

Instances of the PluggableButtonMorph class can be created with the #model:action:label: class method. The model: keyword specifies a model object, the action: keyword specifies a selector for the model object, and the label: keyword specifies the text that appears in the button.

Let's demonstrate using a PluggableButtonMorph by defining a class that opens a window containing a single button. Initially the window background color is white. Clicking the button toggles the color between red and blue.

Create the following class:

```
Object subclass: #ButtonDemo
  instanceVariableNames: 'layout'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Demo'
```

Define the following instance methods:

```
initialize
    | button extent window |
    button := PluggableButtonMorph
        model: self
            action: #toggleColor
            label: 'Toggle'.
    button setBalloonText: 'toggle background color'.
    window := SystemWindow new.
    window setLabel: 'Button Demo'; addMorph: button.

    extent := button morphExtent.
    "Add some space to the left and right of the button label."
    button morphExtent: (extent x + 15) @ extent y.

    "Set window size to the smallest height that contains its submorphs."
    layout := window layoutMorph.
    layout padding: 10.
    window
        morphExtent: 300 @ layout minimumExtent y;
        openInWorld.


toggleColor
    layout color: (
        layout color = Color red
            ifTrue: [ Color blue ]
            ifFalse: [ Color red ] )
```

Open a Workspace and evaluate ButtonDemo new. Hover over the "Toggle" button for a second to see its tooltip (a.k.a balloon text). Click the "Toggle" button several times and note how the window background color changes.

4.2 Menu

Cuis-Smalltalk offers several easy-to-use options for menus to inform or to ask the user to make a choice.

4.2.1 Pop Up

The class PopUpMenu provides an easy way to render a dialog that displays information, asks the user for confirmation, or asks the user to select an option. It is similar to the JavaScript DOM functions `alert` and `confirm`.

For example:

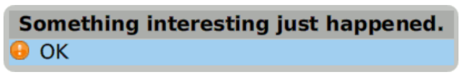


Figure 4.1: A pop up menu to inform

```
PopUpMenu inform: 'Something interesting just happened.'.
```

Example 4.1: Simple pop up menu

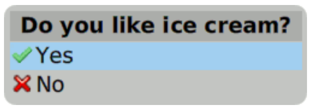


Figure 4.2: A pop up menu to answer Yes or No

```
likesIceCream := PopUpMenu confirm: 'Do you like ice cream?'.
likesIceCream print. "prints true or false"
```

Example 4.2: Yes or No pop up menu

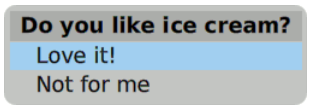


Figure 4.3: A pop up menu to select among two choices

```
likesIceCream := PopUpMenu
    confirm: 'Do you like ice cream?'
    trueChoice: 'Love it!'
    falseChoice: 'Not for me'.
likesIceCream print. "prints true or false"
```

Example 4.3: Two choices pop up menu

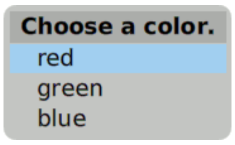


Figure 4.4: A pop up menu to select among several choices

```
color := PopUpMenu withCaption: 'Choose a color.' chooseFrom: #('red' 'green' 'blue').
color print. "prints choice index 1, 2, or 3"
```

Example 4.4: Many choices pop up menu

4.2.2 Selection menu

The SelectionMenu class is a subclass of PopupMenu; it gives a bit more flexibility to the developer. Indeed, once the user selects a menu entry, instead of returning this index entry as PopUp Menu does, it returns an associated object to this entry. It is therefore more flexible.

For example:

```
labels := #('Red sky at sunset' 'A Clockwork Orange'
   'Yellow submarine' 'Green peace' 'The Blue dot' 'Purple rain').
lines := #(3 6). "draw lines after these indexes"
menu := SelectionMenu labels: labels lines: lines.
selection := menu startUpMenu.
selection print. "prints the selected menu entry index"
```

Example 4.5: Selection menu, index answer

Still returns the index of the selected menu entry, which may not be very helpful.

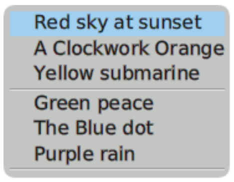


Figure 4.5: A selection menu without title

What we want is a color object instead of an index. To do so, we tell SelectionMenu about a collection of colors from which to obtain the returned value depending on the user-selected menu entry.

```
...
colors := {Color red . Color orange. Color yellow . Color green . Color blue . Color purple}.
menu := SelectionMenu labels: labels lines: lines selections: colors.
selection := menu startUpMenu.
selection print. "prints the selected color"
```

Example 4.6: Selection menu, value answer

In the following example, we demonstrate a use case of SelectionMenu in a MenuDemo class.

```
Object subclass: #MenuDemo
   instanceVariableNames: 'colorButton statusLabel window'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Demos'
```

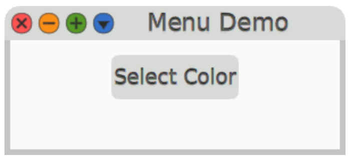


Figure 4.6: A button to invoke a selection menu

We put in place all the widgetery involved.

```
initialize
    colorButton := PluggableButtonMorph
       model: self
       action: #openMenu
       label: 'Select Color'.
    statusLabel := LabelMorph contents: ''.

    window := SystemWindow new.
    window layoutMorph separation: 10.
    window
       setLabel: 'Menu Demo';
       addMorph: colorButton;
       addMorph: statusLabel;
       openInWorld;
       morphExtent: (300 @ window layoutMorph minimumExtent y)
```

Once the button is clicked, we request at runtime the menu with the appropriate selection of colors from which to pick an answer.

```
colorMenu
    ↑ SelectionMenu
       labels: #('Red sky at sunset' 'A Clockwork Orange' 'Yellow submarine' 'Green peace'
          'The Blue dot' 'Purple rain')
      selections: {Color red . Color orange. Color yellow . Color green . Color blue . Color purple}
```

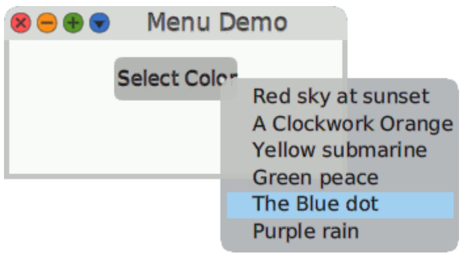
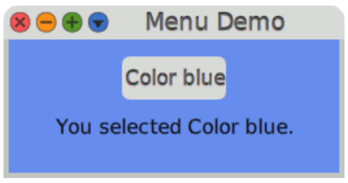


Figure 4.7: A selection menu open

Then we adjust the window color and button label accordingly:

```
openMenu
    | selectedColor |
    (selectedColor := self colorMenu startUpMenu) ifNotNil: [
        colorButton label: selectedColor name.
        statusLabel contents: ('You selected {1}.' format: { selectedColor name }).
        window layoutMorph color: (selectedColor alpha: 0.6) ]
```



You selected Color blue.

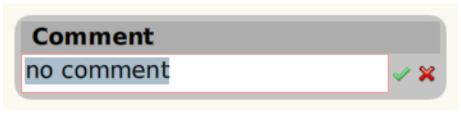
Figure 4.8: Result of the user selection in the menu

To run this, evaluate MenuDemo new in a Workspace.

4.2.3 String Request

The class StringRequestMorph prompts the user to enter a text response. It can verify the response using a provided block that returns a Boolean value indicating whether the response is valid. It can also evaluate a block if the user clicks the cancel button.

For example:



no comment

Figure 4.9: A StringRequestMorph open

```
StringRequestMorph
    request: 'Comment'
    initialAnswer: 'no comment'
    verifying: [ :answer | answer isEmpty not ]
    do: [ :answer | answer print ]
    orCancel: [ 'canceled' print ].
```

Example 4.7: Request a string from the user

4.3 Text Entry

The TextModelMorph class can be used for single or multiple line text input. It is a very important and complex class with a lot of features:

- **Input field.** Single line or multiple lines.

- **Code editing.** Syntax formatting and coloring, code completion.
- **Style.** Text colors and attributes as bold, italic, underlined, etc.
- **Font.** True type font selection with various sizes.
- **Paragraph.** Text alignment.

The following code creates an instance of TextModelMorph, changes its value, and prints its value:

```
textEntry := TextModelMorph withText: 'initial content'.
textEntry openInWorld.
textEntry editor actualContents: 'new content'.
textEntry text print
```

Example 4.8: Simple text entry

Typically, you will want to modify many aspects of this morph such as its size and whether words automatically wrap to new lines. By default, words that would extend past the right side wrap to the next line. To prevent wrapping:

```
textEntry wrapFlag: false
```

The value associated with a TextModelMorph is either held in a TextModel object or a Text object.

There are three common ways to create an instance of TextModelMorph. The choice is based on how the initial value is supplied and where the current value is held.

1. TextModelMorph withText: aTextOrString

   This creates a TextModel object that is initialized with the given value, passes it to the next method, and returns what that returns.
2. TextModelMorph withModel: aTextModel

   The approach allows the same TextModel to be used by multiple other morphs. It creates an instance, sets its model to a TextModel, and returns the instance.

   With this approach, when a TextModel sees its contents changed, it informs its view about the change:

   ```
   actualContents: aTextOrString
      self basicActualContents: aTextOrString.
      self changed: #actualContents
   ```

   And a TextModelMorph listens to such changes to update itself:

   ```
   update: aSymbol
      super update: aSymbol.
      aSymbol ifNil: [ ↑self].
      . . .
      aSymbol == #actualContents ifTrue: [ ↑self updateActualContents ].
      . . .
   ```

3. TextModelMorph textProvider: aTextProvider textGetter: getTextSel textSetter: setTextSel

   This approach allows the value to be maintained in any object, aTextProvider, that responds to the given selectors to provide the text with its *getter* and to update its text attribute from the text entry with its *setter*.

   Underneath, it wraps the text provider in a PluggableTextModel, which is then the model of the text entry[1].

   When a change is notified by the text provider

   ```
   TextProvider >> aMethodDoingSomething
      . . .
      self changed: #getterSymbol.
      . . .
   ```

   The pluggable model receives an update which in turn notifies of the change:

   ```
   PluggableTextModel >> update: aSymbol
      . . .
      aSymbol == textGetter ifTrue: [ ↑self changed: #acceptedContents ].
      aSymbol == selectionGetter ifTrue: [ ↑self changed: #initialSelection ].
      self changed: aSymbol
   ```

   Propagated to the text entry to update itself (method partial shown above):

   ```
   update: aSymbol
      . . .
      aSymbol == #acceptedContents ifTrue: [ ↑self updateAcceptedContents ].
      . . .
   ```

   Beware, the behavior of updateAcceptedContents may be altered by the status of hasUnacceptedEdits attributes. When true, it may prevent the update of the text entry from other mean when the user has previously key-in text. In case of doubt, try to set this attribute to false: textEntry hasUnacceptedEdits: false.

To display prompting text inside a TextModelMorph until the user begins typing a value, send it the message #emptyTextDisplayMessage: with a string argument.

The default background color of a TextModelMorph is white. A TextModelMorph only displays a border when it has focus. One way to make its bounds apparent when it doesn't have focus is to set the background color of the parent component.

```
textEntry owner color: (Color blue alpha: 0.1)
```

Another way is to set the background color of the TextModelMorph.

```
textEntry color: (Color blue alpha: 0.1)
```

[1] For real, the TextModelMorph instance, but the term text entry will be preferably used to avoid confusion.

By default, when there are more lines than fit in the height, a vertical scroll bar appears. When wrapping is turned off, if the text does not fit in the width, then a horizontal scroll bar appears.

To prevent scroll bars from appearing, send the following message to an instance #hideScrollBarsIndefinitely.

The default size of a TextModelMorph is 200 by 100. This is set in the initialize method of PluggableMorph, which is the superclass of PluggableScrollPane, which is the superclass of TextModelMorph. Depending on the font, the default size displays around four lines of wrapping text with around 17 characters per line.

To change the size:

```
textEntry morphExtent: width @ height
```

The size should include space for scroll bars if they may be needed.

Setting the height to zero causes it to actually be set to the height required for a single line in the current font.

By default, pressing the tab key does not move focus from one TextModelMorph instance to another. To enable this, do the following for each instance:

```
textEntry tabToFocus: true
```

When the user changes the text in a TextModelMorph, the object that holds its value is not automatically updated. To manually request the update:

```
textEntry scroller acceptContents
```

There are multiple ways to configure user actions to trigger an update in the model, a text provider or a text model as described previously. The easiest are:

```
textEntry acceptOnAny: true. "updates after every keystroke"
textEntry acceptOnCR: true. "updates after return key is pressed"
```

To listen for changes to the value of a TextModelMorph:

```
textEntry keystrokeAction: [:event |
    | value |
    value := textEntry text.
    "Do something with the value."]
```

By default, if the user attempts to close a SystemWindow and it contains TextModelMorph instances that have unsaved changes, they are asked to confirm this with the message "Changes have not been saved. Is it OK to cancel those changes?" The user must select "Yes" to close the window.

To disable this check for a particular instance of TextModelMorph, send it #askBeforeDiscardingEdits: with the argument false.

The following code creates a single-line text input with a given width that never shows scroll bars:

```
textEntry := TextModelMorph withText: '' ::
    hideScrollBarsIndefinitely;
    morphExtent: 200 @ 0; "calculates required height for one line"
    wrapFlag: false.
```

If the text exceeds the width, use the left and right arrow keys to scroll the text.

To select all the content in an instance, send it #selectAll. To select content from one index to another where both are inclusive:

```
textEntry selectFrom: startIndex to: endIndex.
```

To place the text cursor at the end of the current content:

```
index := textEntry text size + 1.
textEntry selectFrom: index to: index.
```

The following code demonstrates listening for key events. It prints their ASCII codes and character representations to the Transcript.

```
textEntry keystrokeAction: [:evt |
   evt keyValue print.
   evt keyCharacter print ]
```

Let's combine what we have learned above to create a small application. The user can enter their first and last name. Clicking the "Greet" button displays a greeting message below the button.

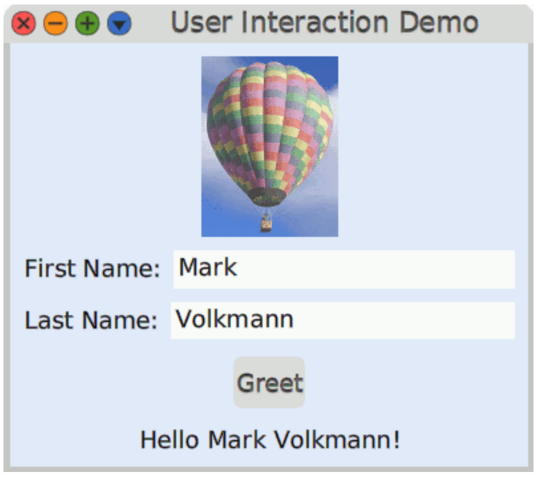


Figure 4.10: User Interaction Demo

Create the class UserInteractionDemo as follows:

```
Object subclass: #UserInteractionDemo
    instanceVariableNames: 'firstName greetLabel lastName'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Demo'
```

Define the following accessor methods:

```
firstName
    ↑ firstName ifNil: [firstName := '']
```

```
firstName: aString
    firstName := aString
```

. . .

As we need two input fields, we define the instance method textEntryOn:

```
textEntryOn: aSymbol
    "Answer a TextModelMorph where aSymbol provides the name for the getter and setter."
    | entry |
    entry := TextModelMorph textProvider: self textGetter: aSymbol textSetter: (aSym-
    bol, ':') asSymbol ::
       acceptOnAny: true; " The model is updated at each key stroke "
       askBeforeDiscardingEdits: false;
       hideScrollBarsIndefinitely;
       " Height to zero causes it to use minimum height for one line. "
       morphExtent: 0 @ 0;
       tabToFocus: true;
       wrapFlag: false.
    entry layoutSpec proportionalWidth: 1. "width is 100%"
    ↑ entry
```

Our text entries are then packed in a row with the appropriate label:

```
rowLabeled: aString for: aMorph
    "Answer a row LayoutMorph containing a LabelMorph and a given morph."
    | row |
    row := LayoutMorph newRow ::
       gap: 10;
       addMorph: (LabelMorph contents: aString);
       addMorph: aMorph.
    row layoutSpec proportionalHeight: 0.
    ↑ row
```

We define an action method for the unique button of our small application:

```
greet
   | greeting |
   greeting := firstName
      ifEmpty: ['']
      ifNotEmpty: [ 'Hello {1} {2}!' format: {firstName. lastName} ].
   greetLabel contents: greeting
```

Finally, our initialize method is packing all the involved morphs together:

```
initialize
   | button image window |

   "Relative file paths start from the Cuis-Smalltalk-Dev-UserFiles directory."
   image := ImageMorph newWith: (Form fromFileNamed: './hot-air-balloon.png' ::
      magnifyBy: 0.5).
   button := PluggableButtonMorph model: self action: #greet label: 'Greet'.
   greetLabel := LabelMorph contents: ''.

   window := SystemWindow new ::
      setLabel: 'User Interaction Demo';
      addMorph: image;
      addMorph: (self rowLabeled: 'First Name:' for: (self textEntryOn: #firstName));
      addMorph: (self rowLabeled: 'Last Name:' for: (self textEntryOn: #lastName));
      addMorph: button;
      addMorph: greetLabel;
      openInWorld.

   " Once the window is open and properly layed out,  we adjust its size
     to the smallest height that contains its submorphs."
   window morphExtent: 400 @ window minimumExtent y.

   " Override the automatic Window color scheme "
   window layoutMorph
      separation: 10;
      color: (Color blue alpha: 0.1)
```

In Chapter 6 [Which components? Where to find more?], page 52, you will learn how to use additional Cuis-Smalltalk packages to ease the creation of such dialog windows.

4.4 List

The PluggableListMorph displays a scrollable list of items. Users can select an item by clicking it or by typing its first few letters.

Let's create a small application that allows users to select a color from a list to change the background color of the window. Users can also add new colors and delete existing colors.

In addition to demonstrating the use of PluggableListMorph, we will also see how to disable buttons when their use is not appropriate.

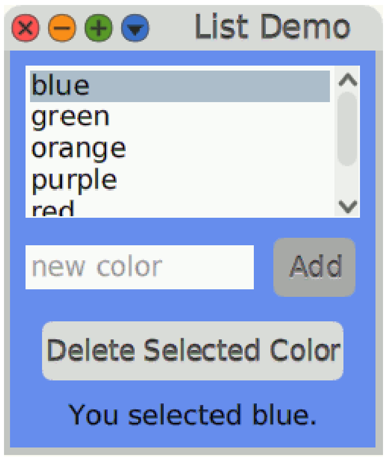


Figure 4.11: List Demo

Create the following class:

```
Object subclass: #ListDemo
    instanceVariableNames: 'colorList colors deleteButton
        newColorEntry selectedColorIndex selectedLabel window'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Demos'
```

Add the following instance methods. We start with a bit long initialize method responsible for building the whole GUI.

```
initialize
   | addButton layout row |

   colors := SortedCollection newFrom: #(red orange yellow green blue purple).
   selectedColorIndex := 0.
   colorList := PluggableListMorph
      withModel: self
      listGetter: #colors
      indexGetter: #selectedColorIndex
      indexSetter: #selectedColorIndex:.
   colorList layoutSpec proportionalWidth: 1.

   newColorEntry :=  self textEntryOn: #newColor.
   newColorEntry emptyTextDisplayMessage: 'new color'.

   addButton := PluggableButtonMorph model: self action: #addColor label: 'Add'.

   row := LayoutMorph newRow
      gap: 10;
      addMorph: newColorEntry;
      addMorph: addButton.

   deleteButton := PluggableButtonMorph
      model: self
      action: #deleteColor
      label: 'Delete Selected Color'.

   selectedLabel := LabelMorph contents: ''.
   window := SystemWindow new.
   window
      setLabel: 'List Demo';
      addMorph: colorList;
      addMorph: row;
      addMorph: deleteButton;
      addMorph: selectedLabel;
      openInWorld.

   "sets initial background color"
   self selectedColorIndex: 0.

   "Set window size to the smallest height that contains its submorphs."
   layout := window layoutMorph.
   layout separation: 10.
   window morphExtent: 250  layout minimumExtent y
```

Observe at the beginning of the initialize method the creation of the list, its model is self, therefore an instance of ListDemo. In a real application, it should be a model object representing some type of data. Associated with the model are the messages to send to self to retrieve the list contents (#colors), to get the index of the selected entry in the list (#selectedColorIndex) and to set the selected index (#selectedColorIndex:).

The methods responding to these messages are implemented in ListDemo:

```
addColor
   self newColor: newColorEntry text
```

```
colors
   ↑ colors
```

```
deleteColor
   selectedColorIndex = 0 ifFalse: [
      colors removeAt: selectedColorIndex.
      self selectedColorIndex: 0.
      colorList updateList.
      selectedLabel contents: '' ]
```

```
newColor
   "In this app there is no need to retrieve this value or even hold it in an instance variable,
    but TextModelMorph requires that this method exists."

   ↑ ''
```

When a new color is input by the user, we add it to the colors collection, update the list, then select this color from the list. To prepare for any additional new color input, we clear the text entry by emitting the event #newColor.

```
newColor: aText
   | potentialColor |

   potentialColor := aText asString withBlanksTrimmed.
   potentialColor ifNotEmpty: [
      colors add: potentialColor asSymbol.
      colorList updateList.
      self selectedColorIndex: (colors indexOf: potentialColor )
      self changed: #clearUserEdits.
      self changed: #newColor ]
```

Remember, in Smalltalk, the index of a collection naturally starts with one. Therefore, an index of zero naturally indicates that no item is selected in the collection.

```
selectedColorIndex
   ↑ selectedColorIndex
```

This method is called when the user clicks or unclicks an item of the list. As recalled earlier, an index of zero indicates that no item was selected. If the color cannot be determined by the name added to the list or is empty, a default gray color is set as the background color of the window.

```
selectedColorIndex: anIndex
   | color colorName selected |

   selectedColorIndex := anIndex.

   selected := anIndex ~= 0.
   deleteButton enable: selected.
   colorName := selected ifTrue: [ colors at: anIndex ].

   selectedLabel contents: (colorName
      ifNil: ['']
      ifNotNil: [ 'You selected {1}.' format: { colorName } ] ).

   color := colorName
      ifNil: [ Color gray ]
     ifNotNil: [ [ Color perform: colorName ] on: MessageNotUnderstood do: [ Color gray ] ].
   window layoutMorph color: (color alpha: 0.6)



textEntryOn: aGetter
   "Answer a TextModelMorph where aGetter provides the symbol for the getter."
   | entry |

   entry := TextModelMorph
      textProvider: self textGetter: aGetter textSetter: (aGetter, ':') asSymbol ::
      acceptOnCR: true;
      askBeforeDiscardingEdits: false;
      hideScrollBarsIndefinitely.

   entry morphExtent: 0 @ 0.
   entry layoutSpec proportionalWidth: 1.
   ↑ entry
```

4.5 Memory Game

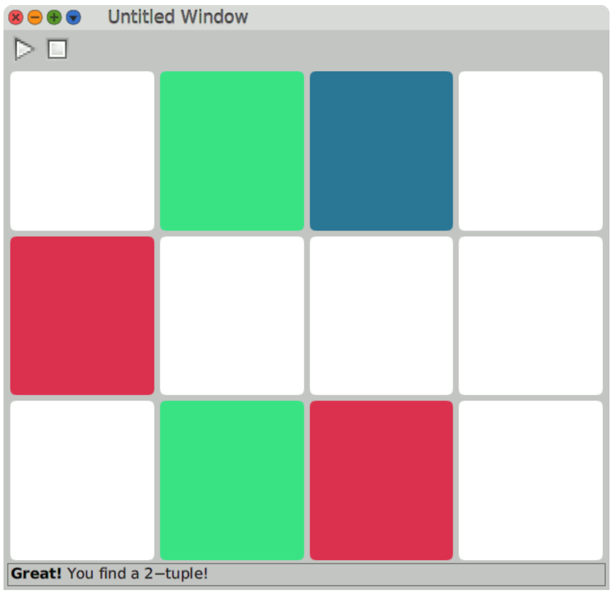
To illustrate user interactions with the mouse, we present in this section a memory color game. This is a board game where a collection of cards are presented to the user with a common neutral color; each card has its own color, hidden at game startup. The user must find the cards sharing the same color. When the user clicks on a card, its color is revealed; the card with the matching color must be found. When a pair of cards are found, these cards are not playable anymore; if not, the cards' colors are hidden again.

Two morphs are used in the design of the game: a kind of SystemWindow and a kind of PluggableButtonMorph. The complete source is presented in the appendix Appendix F [Memory Game v1], page 90, of the book. We will not present every part of the code design, but we will focus on the illustrative ones in regard to the topic of this chapter.

The game is started with:

The game is started with

```
MemoryGameWindow new openInWorld
```



Great! You find a 2–tuple!

Figure 4.12: Memory color game

The Card

What are the attributes we want a card to have? We need a card to have a specific *card color* and a status flag to inform if we are *done* with the card.

The card is a morph able to paint itself with a color. It reacts to user clicks to flip itself between its own *card color* and the common neutral color. What we want is a kind of button:

```
PluggableButtonMorph subclass: #MemoryCard
   instanceVariableNames: 'cardColor done'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV1'
```

The default color, common to all the cards, is white:

```
defaultColor
   ↑ Color white
```

It knows to be flipped when its cardColor is used as the color of the button:

```
isFlipped
   ↑ color = cardColor
```

And the card can flip between this common color and its own cardColor:

```
flip
   color := self isFlipped ifTrue: [self defaultColor] ifFalse: [cardColor].
   self redrawNeeded
```

color is used to paint the button; when adjusting it, we send the message #redrawNeeded to the button to force its redraw.

The Board

What are the attributes of the game board? It knows about the *playground* set with a specific *size* where the *cards* are presented to the user. It communicates messages through its *status bar* and it knows if the user is *playing* or not.

The game is presented in a window with all these attributes:

```
SystemWindow subclass: #MemoryGameWindow
   instanceVariableNames: 'size cards tupleSize statusBar playground playing'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV1'
```

The board presents a toolbar, its playground, and status bar in the window column layoutMorph:

```
initialize
   super initialize.
   size := 4 @ 3.
   tupleSize := 2.
   playing := true.
   playground := LayoutMorph newColumn.
   self installToolbar.
   self addMorph: playground.
   self installCards.
   self installStatusBar
```

Indeed, by default, a SystemWindow comes with a layout, set as a column. The playground is also a column layout; the cards are arranged into several rows.

Installing the Game

Cuis-Smalltalk does not come with any notion of toolbar, but it is fairly easy to create one with a row layout and buttons:

```
installToolbar
    | toolbar button |
    toolbar := LayoutMorph newRow separation: 2.
    button := PluggableButtonMorph model: self action: #startGame ::
        enableSelector: #isStopped;
        icon: Theme current playIcon;
        borderWidth: 2;
        borderColor: Color black;
        setBalloonText: 'Play the game';
        morphExtent: 32 asPoint.
    toolbar addMorph: button.
    . . .
    self addMorph: toolbar layoutSpec: LayoutSpec new useMorphHeight
```

Observe `enableSelector: #isStopped`; indeed the "Play" button is only active when the game `isStopped`. The same applies with the "Stop" button – not shown here. The toolbar height – row layout – is shrunk to the height of its submorphs; to do so, the toolbar is added to the window with the appropriate specification `LayoutSpec new useMorphHeight`.

The cards are installed in several rows in the `playground`, previously emptied. We remember about each card in a special `cards` array we can access with x and y coordinates, also the position of a card in the playground. The colors to be used are randomly chosen and arranged in:

```
installCards
    | colors  row |
    playground removeAllMorphs.
    cards := Array2D  newSize: size.
    colors := self distributeColors shuffled.
    1 to: size y do: [:y |
        row := LayoutMorph newRow.
        1 to: size x do: [:x |  | card |
            card := MemoryCard model: self action: #flip: actionArgument: x@y.
            card layoutSpec proportionalWidth: 1; proportionalHeight: 1.
            card cardColor: colors removeFirst.
            row addMorph: card.
            cards at: x@y put: card ].
        playground addMorph: row ]
```

We make the card interactive; it is a button. When clicked, the message `#flip:` is sent to the game window with the argument the position in the `cards` array and playground:

```
MemoryCard model: self action: #flip: actionArgument: x@y
```

Game Logic

The core of the game logic is in the `flip:` method. It first flips and locks[2] the clicked card:

[2] The `#lock` message is part of the `Morph` protocol, it suppresses all kind of user interaction with a given morph.

```
flip: position
    | flippedCards |
    (cards at: position) flip; lock.
```

Then it detects if all the flipped cards share the same color. To do so, we do a clever trick of Smalltalk: we collect all the colors of the flipped cards, then convert the collection of colors as a Set instance; all duplicated colors are removed. If the size of the resulting set is not 1, it means cards have different colors. In that case, we inform the user with a message, then unflip and unlock the clicked cards:

```
    flippedCards := self flippedCards.
    (flippedCards collect: [:aCard | aCard cardColor]) asSet size = 1 ifFalse: [
       "Give some time for the player to see the color of this card "
       self message: 'Colors do not match!'.
       self world doOneCycleNow.
       (Delay forSeconds: 1) wait.
       " Color does not match, unflip the flipped cards and unlock "
       flippedCards do: [:aCard | aCard flip; unlock].
       ↑ self]
```

If the colors match, we check if we reach the tupleSize association count – initialized by default to 2, to make pairs of cards. If so, we make the cards to flash and we mark them as done so they can't be played anymore:

```
       flippedCards size = tupleSize ifTrue: [
          " We found a n-tuple! "
          self message: 'Great!' bold, ' You find a ', tupleSize asString, '-tuple!'.
          flippedCards do: #flash.
          flippedCards do: #setDone.
```

At this point of the game logic, in the event of a game win, we inform the user and update the game status:

```
          self isGameWon ifTrue: [
             self message: 'Congatuluation, you finished the game!' bold red.
             playing := false] ]
```

Messages to the User

During the game logic, at several occurrences, we informed the user through messages. The messages are printed in the status bar set at initialization time:

```
installStatusBar
    statusBar := TextParagraphMorph new
        padding: 2;
        color: Color transparent;
        borderWidth: 1;
        borderColor: self borderColor twiceLighter ;
        setHeightOnContent.
    self addMorph: statusBar layoutSpec: LayoutSpec new useMorphHeight.
    self message: 'Welcome to ', 'Memory Game' bold
```

Its companion method to write a new text message just updates the contents of the TextParagraphMorph instance:

```
message: aText
    statusBar contents: aText ;
        redrawNeeded
```

A message sent to the status bar can be more than a plain string; it can be a Text instance with styling attributes. To do so, we send specific messages to a string; for example, 'hello' bold converts the 'hello' string as a Text set with a bold style.

Examples of styling:

```
    'Hello' red bold.
    'Hello ' italic, ' my love' red bold.
```

To discover more messages, browse the method categories `text conversion ...` of the CharacterSequence class.

Access and Test

In the core logic of the game, we accessed the flipped cards in the playground. It is a matter of selecting the cards both *not done* and *flipped*:

```
flippedCards
    ↑ cards elements select: [:aCard | aCard isDone not and: [aCard isFlipped] ]
```

The Array2D instance of the cards variable offers access to its cells with x and y coordinates; however, it does not offer the whole range of the Collection protocol, and particularly the select: method. Nevertheless, its underlying elements attribute is an Array, part of the Collection hierarchy; we use it to get the whole power of the Collection protocol.

We proceed the same to select the done cards:

```
doneCards
    ↑ cards elements select: #isDone
```

And undone cards are selected by a subtracting operation, prone to resist code evolution in the card protocol:

```
undoneCards
   ↑ cards elements asOrderedCollection
      removeAll: self doneCards;
      yourself
```

In the core logic of the game, we test if the game is won; it is a matter of testing if all the cards *are done.* In that case, this count is equal to the number of cards in the game:

```
isGameWon
   ↑ (cards elements select: #isDone) size = (size x * size y)
```

The remaining methods of the game do not require comment; they can be read in the complete source code of the Appendix F [Memory Game v1], page 90.

5 Design a GUI Application

Don't ask whether you can do something, but how to do it.
—*Adele Goldberg*

In the previous chapter, we explained how to handle user interactions on widgets such as buttons and text editors. We illustrated it with a complete memory game; we did not particularly pay attention to how the responsibilities were spread across the various involved objects. In a small project, it does not really matter. However, as soon as you want your project to grow, it may lead to complicated code that is difficult to evolve and maintain.

5.1 Responsibilities

In OOP (Object Oriented Programming), we like each object to be responsible for its assigned business. Of course, we can assign kinds of extended businesses to a few objects as we did in Appendix F [Memory Game v1], page 90, but this is not good design.

A growing project produces legacy code. When the responsibilities of the objects are not properly bounded, it makes this code difficult to understand and maintain. A developer willing to make changes will face various challenges and difficulties: distinguishing how the responsibilities are spread across the involved objects, if any; facing long methods that are hard to understand; replacing one behavior or one class by another one, which may require replacing several behaviors; collaborating with several developers, each working independently on one facet of the project, etc.

A good practice is to design an object with a clearly assigned task, ideally only one per object. Each method of the object should come with meaningful names[1], and each one with a clearly assigned task too. Ideally, a method should not be longer than 10 lines.

With this practice in mind, when it comes to GUI (Graphic User Interface) application development, there is a well-known design pattern: MVC (Model View Controller) or its alternative MVP (Model View Presenter). In this design, the responsibilities are spread on three orthogonal axes with no conceptual overlap.

In the following sections, we present the details of this design applied step by step to the memory game presented in the previous chapter.

5.2 Model View Presenter

MVC[2] and MVP[3] designs are very close. MVC, nowadays widely used in GUI and Web application developments, was invented by the Smalltalk community in the late seventies and early eighties. MVP is a subtle variation where the presenter has more responsibilities than the controller and acts more as a middle-man between the model and view objects.

Let's review the responsibilities of each of the three objects.

- **Model.** This object defines a domain by its intrinsic characteristics, related to the intended goal. A Person model may have attributes like family name, first name, and gender; or in another design, it could have a unique anonymous identifier attribute.

[1] The book *Smalltalk with Style* is worth reading to write good Smalltalk code.

[2] `https://en.wikipedia.org/wiki/Model-view-controller`

[3] `https://en.wikipedia.org/wiki/Model-view-presenter`

The model knows nothing about the view and presenter involved in the design. However, a model modifying itself its attributes should have a way to notify the views to update their representation of the model. This is the role of the dependency mechanism explained later in the chapter.

- **View.** It is responsible for displaying to the user a meaningful view of the model, such as a Person model. The view can be passive to display the attribute of a Person model or interactive to edit the attributes of the Person model. One model may be displayed by one or several different views. *Therefore, a model object has no knowledge of the views* acting on it.
- **Presenter.** This object acts as a middle-man between the model and the view, instantiating and gluing both and acting as the entry point in the application[4].

  The presenter also handles the user actions mediated by the views of a given model. Therefore, when the user edits a text entry, clicks on a button, selects an entry in a menu, or drags a visual object, the event is handled to the presenter. Then it decides, depending on the context and the state of the application, what to do with the event, like updating the state of the application and handling suited data to the model eventually.

Now, let's see how to reshape our memory game to fit it in the MVP design.

5.2.1 Memory Game Model

The model is the isolated object, without knowledge of the presenter and the view[5], it is easier to start from there.

In the previous design of the game, we had two classes, MemoryGameWindow and MemoryCard, acting as view and model. Therefore, we need to extract what is model-related.

Our game involves the domain of a game with cards. We define two models:

- MemoryCardModel. It knows about the intrinsic characteristics of a card in the context of the memory game. In the earlier game design, the MemoryCard view knows about its cardColor and its states done and flipped; the latter one was deduced by a method based on the color attribute of the view. These three characteristics are clearly part of the card model and need to be represented by instance attributes:

  ```
  Object subclass: #MemoryCardModel
     instanceVariableNames: 'flipped done color'
     . . .
  ```

  We add the necessary initialization and methods to externally get the state and to update it.

  ```
  initialize
     done := flipped := false
  ```

  When a card has been successfully associated with cards sharing the same colour, we set it as done, and it can't be played anymore

[4] This is an assumed variation from the view used as the entry point of the traditional approach. Being a middle-man, it makes sense it is instantiated first.

[5] Though several models may know about each other.

```
setDone
    done := true
```

To evaluate the available card to play, we need to know if a card is done or not:

```
isDone
    ↑ done
```

During a player move, we need to know if a card is already flipped. If so, the player can't flip it again:

```
isFlipped
    ↑ flipped
```

- MemoryGameModel. This model defines what a memory game is. The previous MemoryGameWindow had 6 instance variables, a mix of view and domain-related attributes: size, cards, tupleSize, statusBar, playground, and playing. Only the first three attributes are related to the characteristics of the game. We may hesitate on the latter one playing, but it will prove to be more useful in the presenter to determine the state of the application.

```
Object subclass: #MemoryGameModel
  instanceVariableNames: 'size tupleSize cards'
  . . .
```

  - size. A point representing the disposition of the cards. 4@3 are 3 rows of 4 cards.
  - tupleSize. An integer, the number of associated cards to find, by default 2.
  - cards. The collection of MemoryCardModel instances.

  We have the necessary methods to initialize the instance and to create the card models. Compared to its counterpart MemoryGameWindow>>installCards of the previous design, the method installCards here is much simpler and easier to understand because it only instantiates the card models. Separation of responsibility is paying off:

```
initialize
   size := 4 @ 3.
   tupleSize := 2


installCardModels
   | colours |
   cards := Array2D newSize: size.
   colours := self distributeColors.
   1 to: size y do: [:y |
     1 to: size x do: [:x |
       cards
         at: x@y
         put: (MemoryCardModel new color: colours removeFirst) ] ]
```

In this class, we also import, unchanged, the behaviors of MemoryGameWindow fitting the game model: distributeColors, doneCards, flippedCards, undoneCards and isGameWon.

That's it for the models of the game.

5.2.2 Memory Game View

We have defined two model classes, so we may expect to define two view classes. Well, not necessarily. Here we just need to define one view class for the whole game, and we use an existing view of Cuis-Smalltalk for the card model, the PluggableButtonMorph.

We need to reshape MemoryGameWindow to contain only view-related business, first in its attributes then its behaviors. First of all, *a view always knows about its presenter*, it can even know about the model through the mediation of the presenter:

```
presenter: aPresenter
    presenter := aPresenter. self
   model: presenter model
```

It also knows about some other views needed for its internal organisation and regulation:

```
SystemWindow subclass: #MemoryGameWindow
   instanceVariableNames: 'presenter statusBar playground'
   . . .
```

Again, the behavior is stripped down to only view considerations, and the initialize method is shortened.

Installing the toolbar slightly differs:

```
installToolbar
   | toolbar button |
   toolbar := LayoutMorph newRow separation: 2.
   button := PluggableButtonMorph model: presenter action: #startGame ::
      enableSelector: #isStopped;
   . . .
```

The model of the buttons is not anymore the view but the presenter.

Indeed, we explained earlier that it is the presenter's responsibility to handle user events. The actions remain the same, and we can anticipate the related methods will be transferred from the view to the presenter class.

Now, we should look at installing the card views:

```
installCards
   | row size |
   playground removeAllMorphs.
   size := model size.
   1 to: size y do: [:y |
      row := LayoutMorph newRow.
      1 to: size x do: [:x | | cardModel cardView |
         cardModel := model cards at: x@y.
         cardView := PluggableButtonMorph
            model: presenter
            action: #flip:
            actionArgument: x@y.
         . . .
         cardView layoutSpec proportionalWidth: 1; proportionalHeight: 1.
         cardView color: cardModel backColor.
         row addMorph: cardView].
      playground addMorph: row ]
```

It relies on the already instantiated card models; we ask the game model all the card models: model cards.

Observe how we just use a stock PluggableButtonMorph as a view for the card. Indeed, we don't need to specialize its behavior, so we keep it simple. Again, the presenter handles the user click on the card. It should be understood as executing the statement presenter flip: x@y at the event[6].

For clarity, we have presented above a shortened version of the installCards method, without the dependencies between the card models and the card views. The logic of installing the card models then the card views is handled by the presenter, the middle-man. We discuss it in the next section.

5.2.3 Memory Game Presenter

We define a new class MemoryGame as our presenter:

```
Object subclass: #MemoryGame
   instanceVariableNames: 'model view playing'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV2'
```

It acts as the entry point of a GUI application, therefore its name is kept short with no *Presenter* fragment. A new game instance is then invoked by a simple:

```
MemoryGame new
```

As we explained earlier, the presenter instantiates both the model and the view:

[6] The message #model:action:actionArgument: to instantiate a button is extremely confusing in its model: keyword; it is not a model as we discussed earlier but only the receiver of the user action, the *controller* in the sense of the MVC pattern.

```
initialize
   model := MemoryGameModel new.
   view := MemoryGameWindow presenter: self.
   self startGame.
   view openInWorld
```

Observe the game model is attached neither to the view nor to the presenter.

From the initialization, the game then is started:

```
startGame
   model installCardModels.
   view installCards.
   view message: 'Starting a new game' bold green.
   view setLabel: 'P L A Y I N G'.
   playing := true
```

By invoking card models and views installations, it is asked to each object in charge of that business.

We already learned the flip: method, called when the user clicks on a card, is now defined in the presenter. The method is quite similar to the previous iteration, except now we only know about the card model. The associated card view is unknown:

```
flip: position
   | flippedCards |
   (model cards at: position)
      flip;
      triggerEvent: #lock.
   flippedCards := model flippedCards.
   . . .
      " Unflip and unlock the flipped cards "
      flippedCards do: [:aCard |
         aCard flip;
            triggerEvent: #flash;
            triggerEvent: #unlock].
      ↑ self].
   . . .
      " We found a n-tuple! "
      view message: 'Great!' bold, ' You find a ', model tupleSize asString, '-tuple!'.
      flippedCards do: [:aCard | aCard triggerEvent: #flash].
      flippedCards do: #setDone.
   . . .
```

Therefore, to update the state of a card view, a card model triggers events which are propagated to any card view listening to the events. An event is coded as a symbol representing an aspect of the model that changed. The symbol name is arbitrarily chosen to be meaningful. In the flip: method, there are three events:

- lock. It informs the card is locked and it cannot be played anymore.

- flash. It informs the card is revealing itself.
- unlock. It informs the card is unlocked and it can be played again.

When triggering an event, it is additionally possible to pass along a parameter. Observe this feature in the model's flip method to inform about the card color changed:

```
MemoryCardModel>>flip
    | newColor |
    flipped := flipped not.
    newColor := flipped ifTrue: [color ] ifFalse: [self backColor].
    self triggerEvent: #color with: newColor
```

All in all, there are four events triggered by a card model: lock, flash, unlock, and color. How a view can listen to a given event is discussed in the next section.

5.3 The Three Musketeers

The model, the view, and the presenter are tied together. Unlike the three musketeers, which were tied together by friendship and the fight for justice, our three objects are tied together by the dependency mechanism we already discussed in the previous sections.

Earlier, we wrote a model does not know about its view(s). However, how can a view be notified its model changed?

5.3.1 update/change

5.3.2 Update/Change

In the MVC design, a view is added as a dependent object to the model it wants to receive updates from:

```
    model addDependent: aView.
    . . .
    model changed: #color
```

Then the dependent view receives #update: message with an argument exposing the changed aspect in the model. It must handle it appropriately:

```
view>>update: aspect
    aspect == #color ifTrue: [self color: model color]
    . . .
```

A view can stop listening to a model to not receive #update: message anymore:

```
    model removeDependent: aView
```

There are two drawbacks to this design: all the changed aspects in the model are handled in a unique update: method in each listening view. If there are a lot of aspects to handle, the update: becomes cluttered. Moreover, the update is sent to all the views, independently of their interest for a particular aspect. Think about a view not interested in the change of the color aspect of a model; it still receives color updates.

The changed/update mechanism appeared with the MVC design. Looking at the implementation of the changed: method is interesting:

```
Object>>changed: aspectSymbol
   "Receiver changed. The change is denoted by the argument.
   Usually the argument is a Symbol that is part of the observer's change
   protocol. Inform all of the observers.
   Note: In Smalltalk-80 observers were called 'dependents'."

   self
      triggerEvent: #changed:
      with: aspectSymbol
```

Underneath, it is implemented with the trigger event we met in the previous section. Indeed, it is now superseded and implemented with the observer pattern, which offers more flexibility. We discuss it in the next section.

The changes/update mechanism is still largely used in the Morphic widget, therefore it is worth getting acquainted with.

5.3.3 The Observer Pattern

When several views are tied to an identical model, a given view may only be interested to receive updates of a specific aspect of the model.

To do so, we specifically register the aspect a view is interested by:

```
model when: #border send: #adjustBorder to: view.
model when: #color send: #setColor: to: anotherView
```

Then the model triggers events:

```
model triggertEvent: #border.
model triggerEvent: #color with: Color red
```

And the effects on the views are equivalent to the sent messages:

```
view adjustBorder.
anotherView setColor: Color red
```

The sent message is set at the registering time of the event with the message #when:send:to: and the optional parameter is set when triggering the event with the message #triggerEvent:with:.

An additional flexibility of the observer pattern: it is not required to subclass the view to implement a specific method, as it was necessary with the update: method.

In our memory game, this is one reason we don't need to define a card view class; we use the stock PluggableButtonMorph. Observe how we register these events. We previously left out this part (See [MemoryGameWindow>>installCards], page 46):

```
installCards
   ...
   cardView := PluggableButtonMorph model: presenter action: #flip: actionArgument: x@y.
   cardModel
      when: #color send: #color: to: cardView;
      when: #lock send:#lock to: cardView;
      when: #unlock send: #unlock to: cardView;
      when: #flash send: #flash to: cardView.
   cardView layoutSpec proportionalWidth: 1; proportionalHeight: 1.
   ...
```

One last detail: to foster your understanding of the relation between these two event mechanisms, the addDependent: method is also implemented with the observer pattern:

```
Object>>addDependent: anObserver
   "Make the given object one of the receiver's observers (dependents).
   Receivers of this message, i.e. objects being observed, are usually called Models.
   Observing can also be set up sending one of the #when:send:to: or #addDepen-
   dent: messages.
   Models will send themselves #changed:, #changed or one of the #triggerEvent* mes-
   sages to notify possible observers.
   If appropriate, it is best to make Models inherit from ActiveModel, for performance reasons."

   self
      when: #changed:
      send: #update:
      to: anObserver.
   ↑ anObserver
```

See Appendix G [Memory Game v2], page 94, for the complete game code.

We end here our chapter regarding the design of a GUI application. The more you will use this design, the more you will appreciate it, particularly when a project grows.

6 Which Components? Where to Find More?

In this chapter, we present additional morphs useful for building GUIs. Some of these components ease the process of designing GUI dialogs as presented in Chapter 4 [Handle user interaction], page 21. However, many bring additional features, and we will present a selection of those.

6.1 Cuis-Smalltalk-UI

Check that you have the `Cuis-Smalltalk-UI` repository already installed. It comes with several Cuis-Smalltalk packages for GUI building. If not present in your host, clone its repository in your parent directory of your installed `Cuis-Smalltalk-Dev` system:

```
cd yourPath/Cuis-Smalltalk-Dev
cd ..
git clone --depth 1 http://github.com/Cuis-Smalltalk/Cuis-Smalltalk-UI
```

The repository contents are divided into several packages whose classes are listed for reference in the `README` page of the repository. In the following chapter, we install individually the appropriate package when needed.

6.2 Easing GUI Design

Let's start the exploration with a few components to ease the creation of a GUI.

6.2.1 Label That Squeezes

Sometimes, labels may have a tendency to occupy more space than available. This becomes particularly true when you do not control the content of a label when an application is translated into other languages, with a more or less verbose way to express messages or concepts.

The SqueezeLabelMorph tries its best to contract a message into a given amount of characters. It is part of the `UI-Core.pck.st` package. In a Workspace, install it by executing the appropriate command:

```
Feature require: 'UI-Core'
```

This kind of label is set with a minimum number of characters it is willing to display. If that minimum number is lower than the label's content, it will contract the text:

```
(SqueezeLabelMorph
   contents: 'I am a very looong label with maybe not enough place for me'
   minCharsToShow: 20) openInWorld.
```

Example 6.1: Label that squeezes

The content of the label is very long, particularly as we inform the label to accept being squeezed to a minimum of 20 characters. Observe how such a squeezed label reveals its complete content in a balloon text when the pointer is hovering over it.

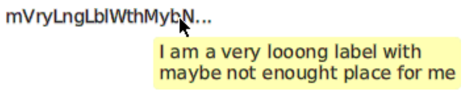

mVryLngLblWthMybN...
I am a very looong label with maybe not enought place for me

Figure 6.1: A label squeezed to 20 characters

When more space is made available to the label, more text of its content is revealed:

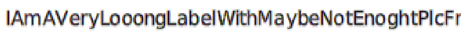


Figure 6.2: A squeezed label given some more space

When packing a SqueezeLabel in a layout morph with other morphs, it will have consequences on the minimal width of the owner layout.

Compare the two examples with a squeezed and regular label:

```
| label row |
label := SqueezeLabelMorph
contents: 'I am a very long label'
minCharsToShow: 15.
row := LayoutMorph newRow.
row
   addMorph: label;
   addMorph: (TextModelMorph withText: 'some input' :: morphExtent: 100@0).
row openInWorld
```

Example 6.2: Squeezed label for a text entry

The whole layout is contracted to a smaller width.

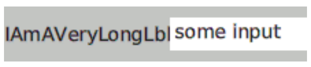


Figure 6.3: A text entry with a squeezed label

When comparing to a regular label use case:

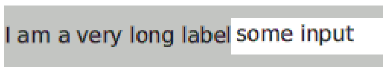


Figure 6.4: A text entry with a regular label

```
| row |
row := LayoutMorph newRow.
row
   addMorph: (LabelMorph contents: 'I am a very long label');
   addMorph: (TextModelMorph withText: 'some input' :: morphExtent: 100@0).
row openInWorld
```

Example 6.3: Regular label for a text entry

It is up to you to decide between the compactness of the GUI and the readability of the labels.

6.2.2 One Line Entry

In Section 4.3 [Text entry], page 26, we presented a quite complex and feature-complete class to handle multiple lines of text editing. When only one line of editing is needed, it is a bit overkill. In that circumstance, you can alternatively use the TextEntryMorph, part of the `UI-Entry` package:

```
Feature require: 'UI-Entry'
```

This class is quite simple, and contrary to the TextModelMorph, it does not need a text model. Therefore, there is no such thing as a changed and update mechanism involved; it is a passive morph.

However, it offers two options to interact with other objects:

1. Send it the message #acceptCancelReceiver: to attach an object answering to the #accept and #cancel messages when the ***Enter*** or ***Esc*** keys are pressed.
2. Send it the message #crAction: to set a block of code, with no argument, to be executed when the ***Enter*** key is pressed.

Let's experiment with the associated object answering to the #accept and #cancel messages. We need a TextEntryDemo class:

```
Object subclass: #TextEntryDemo
   instanceVariableNames: 'value entry'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'DesignGUI-Booklet'
```

At initialization time, we create all the needed objects:

```
initialize
   value := '42'.
   entry := TextEntryMorph contents: value.
   entry acceptCancelReceiver: self.
   entry openInWorld
```

Now we make our TextEntryDemo respond to the #accept and #cancel messages. When pressing *Enter*, we update our value attribute:

```
accept
    value := entry contents.
    'I accepted the input ' print.
    ('value = ', value) print
```

But when pressing *Esc*, we just delete the morph:

```
cancel
    'I discarded the input' print.
    entry delete
```

Observe we only need the accessors #contents/#contents: to set and retrieve its content. It is a very simple class to use. If dependency mechanisms were needed, an intermediate object such as the TextEntryDemo can still be used with the observer pattern.

6.2.3 Labelling Widget

In [example of text entry], page 30, we used layout to associate a text entry with a label. It is something very common when building a GUI. The LabelGroup does exactly that for an arbitrary number of morphs.

```
Feature require: 'UI-Widgets'
```

When creating a LabelGroup, we associate labels and widgets/controls in a unique group. In return, the user gets a layout to be inserted in a dialog or a window.

```
(LabelGroup with: {
'First Name:' -> TextEntryMorph new.
'Last Name:' -> TextEntryMorph new}) openInWorld
```

Example 6.4: Labelling a group of morphs

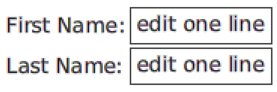
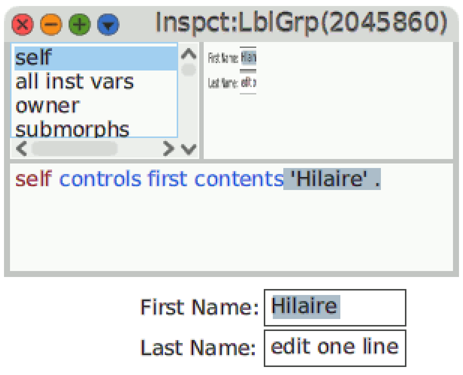


Figure 6.5: Text entries associated with labels

The group also gives access to the controls, although it is not a very efficient way to access the input widgets used in the group; it is handy.



```
self controls first contents 'Hilaire' .
```

First Name: Hilaire
Last Name: edit one line

Figure 6.6: Access to the controls of a label group

A label group is useful when constructing small dialogs. In the next section, we build one with the morphs we learned in this section and the previous ones.

6.2.4 Packing in Panel & Dialog

Small windows the user interacts with are called dialogs or panels. `Cuis-Smalltalk-UI` offers several alternatives to use.

```
Feature require: 'UI-Panel'
```

Let's rewrite the [example of text entry], page 30, with what we just learned. The end result will look like this:

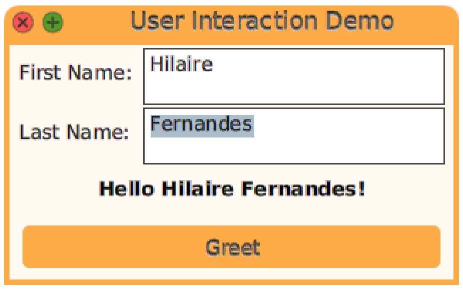


Figure 6.7: A greeting dialog

In the hierarchy provided by the `UI-Panel` package, we use the DialogPanel class. It offers both an area to plug our interactive components and an area for our button.

```
DialogPanel subclass: #GreetingPanel
   instanceVariableNames: 'firstName lastName greetLabel'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'DesignGUI-Booklet'
```

We set the default color of the dialog:

```
defaultColor
   ↑ `Color lightOrange`
```

Then install the iconic buttons for its title:

```
initialize
   super initialize.
   self showButtonsNamed: #(close expand)
```

To know about the available buttons for the title bar of a panel, read the class WindowTitleMorph. The expand action needs a rewrite of its associated action:

```
expandButtonClicked
   self fullScreen
```

We set our components in the dedicated newPane method of the Dialog hierarchy:

```
newPane
   | column group |
   column := LayoutMorph newColumn ::
      color: Color transparent;
      gap: 10 .
   group := LabelGroup with: {
      'First Name: ' -> (firstName := TextEntryMorph contents: '') .
      'Last Name: ' -> (lastName := TextEntryMorph contents: '') }.
   greetLabel := LabelMorph contents: '' font: nil emphasis: 1.
   column
      addMorph: group layoutSpec: (LayoutSpec fixedWidth: 300);
      addMorph: greetLabel layoutSpec: (
         LayoutSpec proportionalWidth: 0 fixedHeight: 20 offAxisEdgeWeight: #center).
   ↑ column
```

The button has its own method too for installation:

```
newButtonArea
   ↑ PluggableButtonMorph model: self action: #greet label: 'Greet' ::
      color: self widgetsColor
```

Finally, we implement the greet action of the button to update the greetLabel:

```
greet
   greetLabel contents: (
      'Hello {1} {2}!' format: {firstName contents. lastName contents})
```

6.3 Additional Features

In the next sections, we present a very small selection of useful components. There are many more to explore in the repository; they all come with example methods to learn from in their class side.

6.3.1 Radio and Check Buttons

These widgets are found in the `UI-Click-Select` package.

In a GUI, a check button represents a boolean value. They are used in groups with a text label or any kind of morph to represent its iconic representation.

```
| column group |
column := LayoutMorph newColumn.
group := CheckGroup fromList: #('Cuis' 'Pharo' 'Squeak').
group buttons do: [:each | each when: #checkSelection send: #show: to: Transcript].
column
   addMorph: (LabelMorph contents: 'I use...' bold);
   addMorph: group.
column openInWorld
```

Example 6.5: Check group

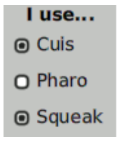


Figure 6.8: A group of check buttons

To meaningfully manage the check and uncheck events, different action methods can be set for each check box. The previous example could be rewritten:

```
. . .
cuisCheck := [:check |
   check isSelected
   ifTrue: ['I use Cuis' print] ifFalse: ['I don''t use Cuis' print] ] .
group buttons first
   when: #checkSelection
   send: #value:
   to: cuisCheck.
. . .
```

Do not add the cuisCheck variable in the declaration at the first line of the script; otherwise, it will be garbage collected. Indeed, action events are weakly referenced – i.e., it does not add a count to its reference use. Of course, in an application, you will use a method selector as the argument of the #to: keyword.

In a radio group, only one check button is selected at a time. When a new button is selected, the previously selected button is deselected. A radio button is drawn differently, as a circle. We can alter our previous greeting dialog to add a radio group to select a preferred color:

```
newPane
   | column group radioColors |
   . . .
   radioColors := RadioGroup fromList: #(Blue White Red).
   radioColors when: #informRadioSelection send: #setColor: to: self.
   column
      addMorph: group layoutSpec: (LayoutSpec  fixedWidth: 300);
      addMorph: radioColors beRow;
   . . .
```

And the associated action method:

```
setColor: aColorSymbol
   self color: (Color perform: aColorSymbol asLowercase asSymbol)
```

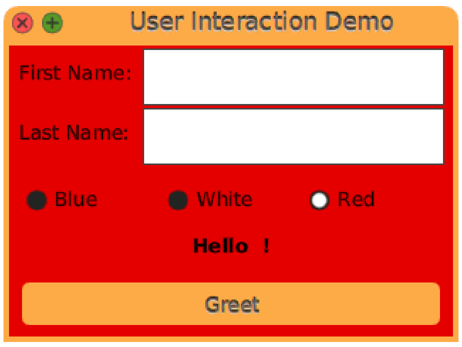


Figure 6.9: Our enhanced dialog to select color with radio buttons

6.3.2 Drop Down Button

A drop-down button is a button whose content is selected from a drop-down list. When the user clicks such a button, a menu is displayed below the button. The content of the list is textual when using the class DropDownListMorph, or it can be any kind of morph when using the class DropDownButtonMorph.

We can add a drop-down button to let the user select an icon to decorate our greeting dialog. We adjust our existing code to add an image morph to the pane and the drop-down button:

```
newPane
    | column group radioColors |
    . . .
      'Last Name: ' -> (lastName := TextEntryMorph contents: '').
      'Preferred icon: ' -> self dropDownIcons }.
    . . .
    column
      addMorph: (myImage := ImageMorph new);
      addMorph: group layoutSpec: (LayoutSpec  fixedWidth: 300);
    . . .
```

The creation of the button is delegated to another method:

```
dropDownIcons
    | morphs listModel |
    morphs := #(addressBookIcon chatIcon clockIcon doItIcon findIcon) collect: [:anIcon |
      ImageMorph new ::
        image: ((Theme current perform: anIcon) magnifyTo: 32@32) ].
    listModel := ListModel with: morphs.
    listModel when: #listSelectionChanged send: #updateIcon: to: self.
    ↑ DropDownButtonMorph
      withModel: listModel listGetter: #list indexGetter: #listIndex indexSetter: #listIndex:
```

We create a collection of image morphs, each with a different icon. We use a ListModel to hold the collection and trigger events when the *list selection is changed*. In that circumstance, the message #updateIcon: is sent to the dialog with the argument the list model; this method is straightforward:

```
updateIcon: aListModel
    myImage image: aListModel selectedItem form
```

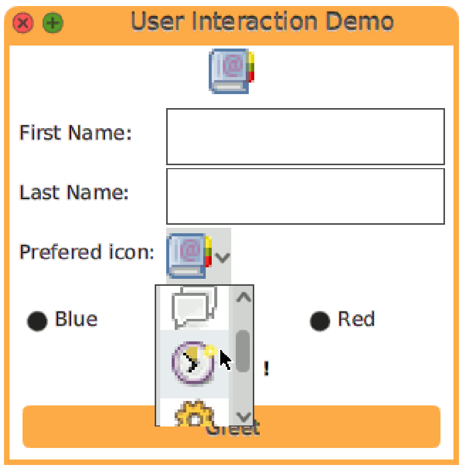


Figure 6.10: A Drop down button of image morphs

It is possible to use an application model instead of a ListModel. In that circumstance, the associated getter and setter would be implemented in this model too.

6.3.3 Decorating Component

Decorating a component is a nice way to set a label around one or several widgets, but it is much more than that. The decorated components are highlighted with a surrounding line and a textual label, then an optional list of quick buttons. The quick buttons can be anything to operate on the surrounded components.

The class DecoratedPane is part of the `UI-Panel`; it is likely already installed on the Cuis-Smalltalk system of the reader if the previous sections of this booklet were read. A decorated pane expects a morph to decorate, a string, and an optional collection of buttons:

```
(DecoratedPane
  open: (Sample03Smiley new)
  label: 'Be Happy') openInWorld
```

Example 6.6: Decorating a morph

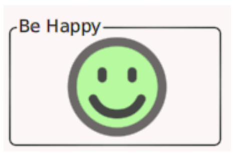


Figure 6.11: A smiley decorated with a 'Be Happy' slogan

Let's go back to our greeting panel and decorate the greetLabel with an information label and two quick buttons: one to reset the greeting and a second one to greet the author of the running Cuis-Smalltalk image. We need to edit again our newPane method:

```
newPane
    | column group radioColors decorator |
    . . .
    greetLabel := LabelMorph contents: '' font: nil emphasis: 1.
    decorator := DecoratedPane
      open: greetLabel
      label: 'Decorated Label'
      quickButtons: {
        PluggableButtonMorph
          model: [greetLabel contents: 'Hello ', Utilities authorName] action: #value ::
          icon: (Theme current fetch: #('16x16' 'actions' 'contact-new')) ;
          setBalloonText: 'Say hello to the Smalltalk author of this running Cuis image.' .
        PluggableButtonMorph
          model: [greetLabel contents: ''] action: #value ::
          icon: (Theme current fetch: #('16x16' 'actions' 'edit-clear'));
          setBalloonText: 'Take back my greeting.' }.
    . . .
```

Then, in the column morph, we add the decorator instead of the greetLabel:

```
newPane
   . . .
   column
      . . .
      addMorph: radioColors beRow;
      addMorph: decorator.
   . . .
```

Observe the PluggableButtonMorph; we use BlockClosure as a model and the message #value as the action to get it executed at button click. In a real application, you will more likely use an instance as a model and an associated method of its protocol.

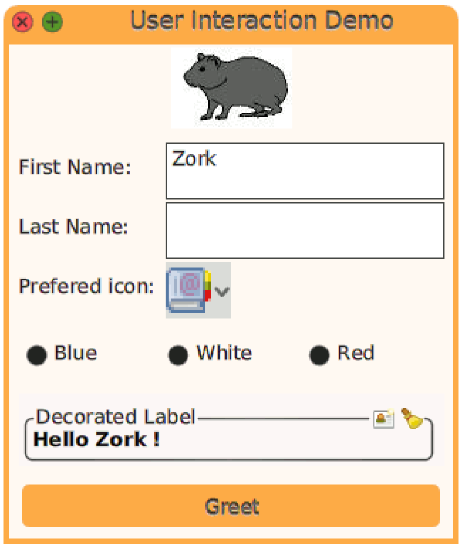


Figure 6.12: The greeting label decorated with two quick buttons

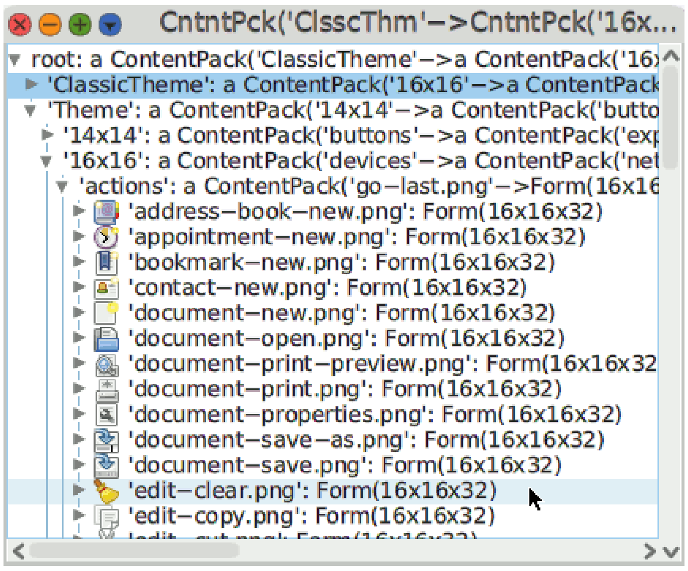
6.3.4 Importing Icons

Icons are important when designing GUIs. Cuis-Smalltalk comes with a few sets of icons. Explore the `icons` method category of the Theme class; each of these methods returns a Form to use as an icon in a PluggableButtonMorph.

```
PluggableButtonMorph
   model: self action: #close ::
   icon: Theme current closeIcon
```

There are additional icons found in ContentPack instances. Invoke an explorer on Theme content[1] to browse those packs:

[1] In a Workspace, highlight Theme content and do *Ctrl-Shift-I*



```
CntntPck('ClsscThm'–>CntntPck('16x...
▼ root: a ContentPack('ClassicTheme'–>a ContentPack('16›
  ▶ 'ClassicTheme': a ContentPack('16x16'–>a ContentPack
  ▼ 'Theme': a ContentPack('14x14'–>a ContentPack('butto
    ▶ '14x14': a ContentPack('buttons'–>a ContentPack('exp
    ▼ '16x16': a ContentPack('devices'–>a ContentPack('net
      ▼ 'actions': a ContentPack('go–last.png'–>Form(16x16
        ▶ 'address–book–new.png': Form(16x16x32)
        ▶ 'appointment–new.png': Form(16x16x32)
        ▶ 'bookmark–new.png': Form(16x16x32)
        ▶ 'contact–new.png': Form(16x16x32)
        ▶ 'document–new.png': Form(16x16x32)
        ▶ 'document–open.png': Form(16x16x32)
        ▶ 'document–print–preview.png': Form(16x16x32
        ▶ 'document–print.png': Form(16x16x32)
        ▶ 'document–properties.png': Form(16x16x32)
        ▶ 'document–save–as.png': Form(16x16x32)
        ▶ 'document–save.png': Form(16x16x32)
        ▶ 'edit–clear.png': Form(16x16x32)
        ▶ 'edit–copy.png': Form(16x16x32)
```

Figure 6.13: Content packs in Cuis-Smalltalk

To use one particular icon of a given content pack, you specify its size, the name of the content pack it belongs to, and the icon name without the file extension:

```
Theme current fetch: #('16x16' 'actions' 'appointment-new')
```

Several icons come with *shortcut* found in the Theme class. Now you may want to use alternative icons; this is where you use the IconImporter class:

```
Feature require: 'UI-Graphic-Import'
```

Beware, it installs the `SVG` package too and its dependencies. It lets you import both PNG and SVG graphic files and scale them at different square sizes. You create an instance with a path where to search for the icons:

```
icons := IconImporter path: '/home/dev/Dynamic-Book/icons'
```

Then you ask for a Form or an ImageMorph of a given icon. To use the file `group.svg` located in `'/home/dev/Dynamic-Book/icons'` as an icon and to scale it as a 32x32 pixels form, you write:

```
icons getForm: #group32
```

Or to request an image morph for direct use in a GUI of your own:

```
(icons getMorph: #group32) openInHand
```

Figure 6.14: Request an ImageMorph from a graphic file

The requested icons of a given size are cached in the icon importer; later requests of the same icon and size have a minimal processing cost.

There are several options to explore in IconImporter, particularly when you need to adjust the icon size to the screen density:

```
" to request an icon with a size set at execution time "
icons getForm: #group ofSize: MyApp iconSize.

" to open an image morph with the given icon "
(icons getMorph: #group ofSize: 64) openInWorld
```

With MyApp deducing the iconSize from user preferences.

When dealing with SVG graphics, monochrome icons may be painted with a given color:

```
icons getForm: #group ofSize: 64 fill: Color red
```

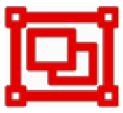


Figure 6.15: Get a picture as an icon and paint it in red

7 Advanced Design Patterns in GUI

In section Chapter 5 [Design a GUI application], page 43, we presented a fundamental design pattern of GUI applications. As the development of your GUI application progresses and grows in complexity, you may want to use additional design patterns. The book *Design Patterns Smalltalk Companion* is a must-read for any serious application development. Design patterns help to develop code that is easier to maintain; they provide a common ground easing mutual understanding.

In the following sections, we present several of these patterns with concrete use cases in applications developed with the Cuis-Smalltalk system.

7.1 Command

Among the design patterns, it is likely one that impacts the most the end-user: it lets you implement the undo and redo operations.

The template repository `CuisApp`[1] demonstrates the design of a picture viewer, as a simple Cuis application example. The user can rotate, flip and scale a picture Its undo and redo actions are implemented with three classes:

- AppCommandStack. Its instance records the history of commands. It is a sort of ReadWriteStream with additional behavior, particularly to truncate the stack of commands. Indeed, when a new command is inserted in the stack – it may not be at the end depending on the user activation of the `undo` action – the tail of the stack must be truncated with the truncate method. A previous method is also needed for the `undo` action.
- AppCommandManager. Its instance manages the stack of commands: create new commands from user actions then undo and redo commands. For each specific user action, there is a dedicated method: flipHorizontally, rotateLeft, zoomIn, etc.

  All these methods share the same pattern: instantiate a command, then execute it:

  ```
  flipHorizontally
     |command|
     command := stack nextPut: (AppFlipHCommand presenter: presenter).
     ↑ command execute
  ```

  Some commands require additional parameters:

  ```
  rotateLeft
     |command|
     command := stack nextPut: (AppRotateCommand presenter: presenter).
     command degrees: -90.
     ↑ command execute
  ```

- AppCommand. The top-level class of a hierarchy of commands, one for each user operation. There are the subclasses AppFlipHCommand, AppRotateCommand, etc. Each of these classes implements the execute and unexecute methods differently.

[1] `https://github.com/hilaire/CuisApp`

For some actions like flip horizontally, the unexecute is identical to execute:

```
AppFlipHCommand>>execute
   self imageMorph image: (self imageMorph form flippedBy: #vertical)
```

```
AppFlipHCommand>>unexecute
   self execute
```

Rotating requires an additional behavior (rotatedBy: not copied here):

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

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

The execute and unexecute methods void their effects. In some circumstances, the execute may have a destructive effect as it happens in the zoom out action; an additional attribute is then used to keep a copy of the initial data:

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

So undoing restores properly the data:

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

How execute and unexecute are implemented really depends on the nature of the command.

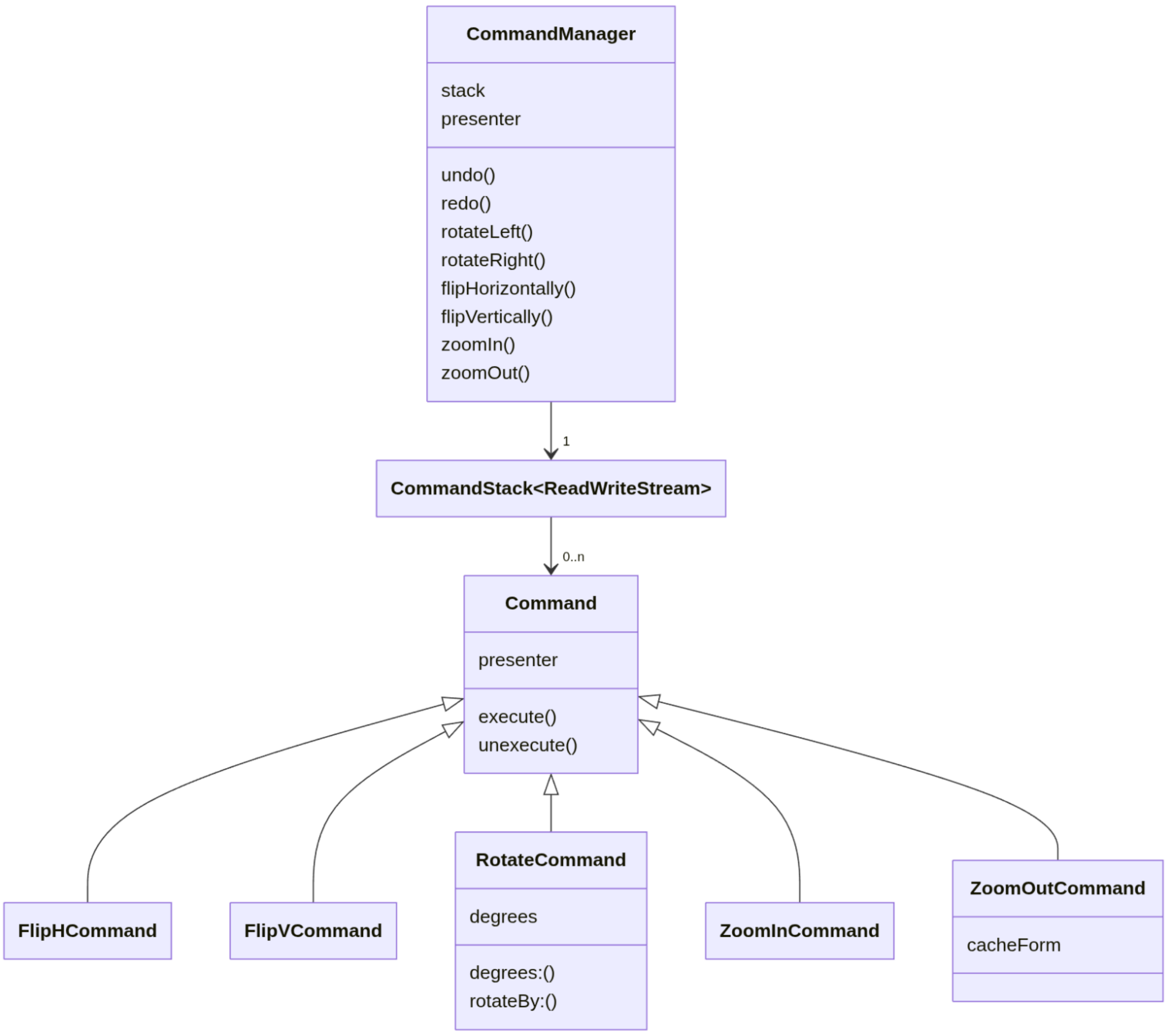


Figure 7.1: Command pattern diagram (for ease of reading the *App* prefix is removed)

7.1.1 Memory Game with Undo/Redo

What will be the implication to implement the undo and redo actions within our game? Our game is simple; the user can only click one card at a time. Therefore, we want to record this and also the consequences on the game state, if any. The consequences can be none, matching cards, or non-matching cards; each resulting in a different changed game state.

When implementing the user interaction with a command, the `execute` method will deal with these three possible outcomes. Its `unexecute` counterpart will have to reverse the game to its original state.

If the game state has a small memory footprint, it is less cumbersome to just save its state before executing each user action. The game state is the collection of each card's status: the done and flipped Boolean values.

Our Memory game just needs to be flanked with the command classes we described earlier, unchanged. Then the Command hierarchy will have one subclass PlayCardCommand:

```
Command subclass: #PlayCardCommand
  tanceVariableNames: 'status position'
  . . .
```

It captures the game state in its status attribute, of the same nature as the cards array in the Memory game model:

```
initialize
   status := Array2D newSize: presenter model cards size
```

At command execution,

```
execute
   self backupModels.
   presenter flip: position
```

the game state is backed up before flipping the card:

```
backupModels
   | size |
   size := presenter model cards size.
   1 to: size y do: [:y |
     1 to: size x do: [:x | | card |
       card := presenter model cards at: x@y.
       status at: x@y put: (Array with: card isFlipped with: card isDone) ]]
```

The undo action restores the game state before execution:

```
unexecute
   " Restore the status of the card models "
   | size |
   size := status size.
   1 to: size y do: [:y |
     1 to: size x do: [:x | | cardStatus card |
       card := presenter model cards at: x@y.
       cardStatus := status at: x@y.
       card
         flip: cardStatus first;
         done: cardStatus second ] ]
```

The card models' flip: and done: methods are refactored to trigger events propagated to the card view:

```
MemoryCardModel>>flip: boolean
    " Set my flip state and trigger a color event for my view accordingly to my flip state "
    | newColor |
    flipped = boolean ifTrue: [↑ self].
    flipped := boolean.
    newColor := flipped ifTrue: [color] ifFalse: [self backColor].
    self triggerEvent: #color with: newColor
```

and

```
done: boolean
    done = boolean ifTrue: [↑ self].
    done := boolean.
    self triggerEvent: (done ifTrue: [#lock] ifFalse: [#unlock])
```

In Appendix H [Memory Game v3], page 99, you will find the complete source of the modified Memory game: toolbar with undo and redo buttons.

7.2 Flyweight

The idea of the flyweight pattern is to have objects in one place and to avoid duplication. It is often associated with the factory pattern, so this will be both the place where objects are manufactured or retrieved.

DrGeo uses the flyweight and factory patterns to manage the geometric objects created by the user. A geometric object is manufactured once the user provided enough information by selecting a set of existing objects in a sketch.

To create a segment AB, the user selects point A and point B. If the user creates a new segment and selects again the point A and the point B, a new segment is not created but instead the existing segment AB is answered. Identically, if the user selects point B then point A, the same existing segment AB is answered. This mechanism occurs in the flyweight factory.

In DrGeo, there are three classes. An abstract DrGFactory:

```
Object subclass: #DrGFactory
   instanceVariableNames: 'pool last'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'DrGeo-Factories'
```

Then two subclasses, DrGMacroFactory and DrGMathItem, to manage the pool of macro-constructions and mathematics items.

The existing objects are held in a pool of objects:

```
initialize
    pool := OrderedCollection new
```

when a new object is pushed in the factory[2], it is first searched in the pool:

```
pushAsLastWhenInPool: anItem
   "If this item has a twin in the pool, push as last this last one and return true,
   otherwise return false"

   ↑ (self findInPool: anItem)
      ifNotNil: [ :item |
         self last: item.
         true ]
      ifNil: [ false ]
```

To find in the pool, its index, if any, is searched:

```
findInPool: item
   "Try to find a twin of this mathItem, if so return the twin, otherwise nil"
   ↑ self at: (self indexOf: item in: pool)
```

Determining if an object has a twin depends on the nature of each object. It is done through hash and equality check at the object level:

```
indexOf: anItem in: aPool
   "No identity equality but hashed value to detect duplicated object we must con-
   sider as equal"
   ↑ anItem
      ifNil: [0]
      ifNotNil: [aPool findFirst: [ :each |
         each hash = anItem hash
         "double check when hash is equal (can be a collision)"
         and: [each = anItem] ] ]
```

If we look at our segment AB example, we want to establish segment AB and segment BA as the same mathematics object.

In DrGeo, the geometric object model uses the template method pattern to establish how equality is articulated:

```
DrGMathItem>>= aMathItem
   ↑ aMathItem isMathItem
      and: [self basicType == aMathItem basicType
      and: [self nodeType == aMathItem nodeType
      and: [self parentsEqual: aMathItem] ] ]
```

For a segment defined by two points, basicType is #segment and nodeType is #'2pts'.

The subclasses of DrGMathItem implement the parentsEqual:. In DrGSegment2ptsItem, we cross-check if the extremities of the segments are the same points:

[2] The item are not built in the factory, this process is factored out in a familly of builder classes discussed in the next section.

```
parentsEqual: aMathItem
   ↑ parents asSet = aMathItem parents asSet
```

Comparing can be slow, therefore we compare first the hash value of each object, pre-computed:

```
rehash
   ↑ hash := (parents asSet hash
   bitXor: self nodeType hash)
   bitXor: self basicType hash
```

These details show the tricky part in this pattern: how to establish two objects are identical. With objects representing mathematics items, it is difficult; the identity check done by DrGeo is very limited. After all, we could have two segments mathematically identical through a process of transformations, and only a mathematics solver could establish it.

7.3 Builder

A builder is an object specialized to construct one or several types of objects. The type of the constructed object depends on the nature of the inputs sent to the builder. Explained differently, depending on input received by the builder, it will figure out what and how to construct a new object with the provided inputs.

In DrGeo, there is a whole hierarchy of 39 builders. The abstract class DrGMathItemBuilder provides the general mechanism through template methods. When an object is added to the builder, first it is asked if it is wanted or not. Next, once the object is added to the builder, it is asked if the builder is ready to build a new object. If so, an event is propagated to whatever is listening:

```
add: aMathItemCollection at: aPoint
   "
   Add a math item in the selection list of the builder, aPoint is the position where the ac-
   tion took place. Return true if mathItem accepted
   "
   ↑ (self isWanted: aMathItemCollection at: aPoint)
      and: [
         self addItem: aMathItemCollection at: aPoint.
         "Are we done? If so notify our dependent"
         self readyToBuild ifTrue: [self triggerEvent: #readyToBuild].
         true]
```

Let's look at the subclass DrGBuilderMiddle, a builder to construct the middle of two points or a segment.

```
   DrGMathItemBuilder subclass: #DrGMiddleBuilder
      instanceVariableNames: 'pointA pointB segment'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'DrGeo-Presenter-Builder'
```

The methods are overridden to fit the task. The middle builder filters items which are either point or segment and, depending on the history of the already added item to the builder:

```
isWanted: aMathItemCollection
   ↑ aMathItemCollection notEmpty
     and: [
       (aMathItemCollection first isPointItem and: [aMathItemCollection first ~= pointA])
       or: [aMathItemCollection first isSegmentItem and: [pointA isNil]] ]
```

Observe a segment is not wanted if a point – pointA – was already provided to the builder.

Adding an item to a builder also depends on the history of added items; pointB will be set once pointA is already known:

```
addItem: aMathItemCollection at: aPoint
   super addItem: aMathItemCollection at: aPoint.
   aMathItemCollection first isPointItem
     ifTrue: [
       pointA ifNil: [
         pointA := aMathItemCollection first.
         ↑ self].
       pointB ifNil: [
         pointB := aMathItemCollection first.
         ↑ self ] ]
     ifFalse: [segment := aMathItemCollection first]
```

Once the segment or the point A and B are known, we are ready to build the middle point:

```
readyToBuild
   ↑ segment notNil or: [pointA notNil and: [pointB notNil]]
```

In the DrGeo GUI, each builder is associated with a tool activated by a button or a menu entry. A builder is very handy to both give feedback to the user and to interpret what the user wants to do.

For example with the DrGMiddleBuilder:

- **Feedback.** Thanks to the #isWanted: message, DrGeo selectively shows balloon text on the relevant mathematics items to the builder

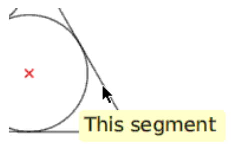


Figure 7.2: A balloon text indicates this segment can be selected

but does not when hovering non relevant mathematics items

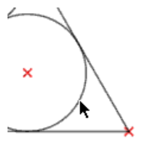


Figure 7.3: No balloon text indicates this circle can't be selected

- **Interpret.** Once a mathematics items is added to the builder, the #isWanted: discards additional options for future selection

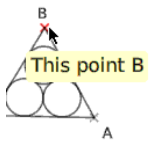


Figure 7.4: Point A was selected, it can't be selected again nor the segment AB, but point B can

7.4 Template Method

This pattern is massively used, very likely it is present in your code. The principle is to write the structure of a method in an abstract class. The method uses auxiliary methods implemented in sub-classes, each one with a different and specific implementation. In the abstract class, these methods will respond self subclassResponsibility or provide a minimal implementation to be overridden in sub-classes.

In Section 7.2 [Flyweight], page 69, we mentioned briefly the template method pattern in a use case to implement equality between two mathematics items:

```
DrGMathItem>>= aMathItem
   ↑ aMathItem isMathItem
      and: [self basicType == aMathItem basicType
      and: [self nodeType == aMathItem nodeType
      and: [self parentsEqual: aMathItem] ] ]
```

Although DrGMathItem provides a simple implementation of parentsEqual:,

```
parentsEqual: aMathItem
   ↑ parents = aMathItem parents
```

many of its subclasses override it to adjust to the mathematics item:

```
DrGAngleBisector3ptsItem>>parentsEqual: aMathItem
   ↑ parents = aMathItem parents
      or: [parents reverse = aMathItem parents]
```

```
DrGPolygonNptsItem>>parentsEqual: aPolygon
   | shiftedCollection |
   parents size = aPolygon parents size ifFalse: [↑ false].
   shiftedCollection := parents.
   shiftedCollection size timesRepeat: [
      shiftedCollection = aPolygon parents ifTrue: [↑ true].
      shiftedCollection := shiftedCollection shiftRight].
   shiftedCollection := parents reverse.
   shiftedCollection size timesRepeat: [
      shiftedCollection = aPolygon parents ifTrue: [↑ true].
      shiftedCollection := shiftedCollection shiftRight].
   ↑ false
```

In the Section 7.3 [Builder], page 71, we have a template method too to decide when to add a mathematics item to a builder:

```
add: aMathItemCollection at: aPoint
   ↑ (self isWanted: aMathItemCollection at: aPoint)
      and: [
         self addItem: aMathItemCollection at: aPoint.
         "Are we done? If so notify our dependent"
         self readyToBuild ifTrue: [self triggerEvent: #readyToBuild].
         true]
```

This time the isWanted: and readyToBuild methods are not implemented in the top class of the builder hierarchy:

```
isWanted: aMathItemCollection
   " Check if the builder is interested by aMathItem "
   self subclassResponsibility
```

```
readyToBuild
   "Can the builder build the math item now?"
   self subclassResponsibility
```

Only subclasses implement the behavior to fit their purpose. For example, in DrGLocus-Builder:

```
isWanted: aMathItemCollection
   ↑ aMathItemCollection notEmpty and: [
      (aMathItemCollection first isPointItemOnCurve and: [freePoint isNil])
      or: [aMathItemCollection first isConstrainedPointItem and: [constrainedPoint isNil]] ]
```

```
readyToBuild
   ↑ freePoint notNil and: [constrainedPoint notNil]
```

buildItem is another template method. It builds an item once enough relevant mathematics items were added to the builder:

```
buildItem
   | itemDefinitions |
   itemDefinitions := self mathItemClass.
   ↑ itemDefinitions isCollection
      ifFalse: [{itemDefinitions newWith: self arguments in: presenter}]
      ifTrue: [point := point + (0.2@1.3).
         itemDefinitions collect: [:class |
            point := point - (0@0.5).
            class newWith: self arguments in: presenter]]
```

Its abstract methods mathItemClass and arguments are implemented in the subclasses. Again, for locus:

```
mathItemClass
   ↑ DrGLocus2ptsItem
```

```
arguments
   ↑ {freePoint. constrainedPoint}
```

The builder pattern is very handy to deal with the complexity of constructing various kinds of objects, each one with its own constraints. In DrGeo, a builder is associated with a tool and state to manage its input and its progress toward creating the new mathematics item. We present tool and state in the next section.

7.5 State

In DrGeo, when the user clicks on a button to operate on the sketch, it is selecting a tool: to move items, to edit the styles, to delete items, to create new ones, etc.

A tool is often associated with a builder, and it can handle different states through dedicated methods. Possible states are: handleChoice:, handleMouseAt:, handlePress:, handlePressShiftKey:, handleRelease, handleReleaseShiftKey, handleShiftKey:, handleShiftKeyMouseAt:, handleStillPress:

The DrGDrawable class, a Morph used as a canvas for the sketch, is responsible for the low-level duty of translating user actions with pointer and keyboard to message sending to one of the methods enumerated above.

The messages are sent to the current tool. For example, when the pointer is hovering the sketch:

```
DrGDrawable>>mouseHover: evt localPosition: localEventPosition
   "handle mouse move with button up"
   localEventPosition = prevMousePos ifTrue: [↑ self]. "nothing to process"
   evt shiftPressed
      ifTrue: [self tool handleShiftKeyMouseAt: localEventPosition]
      ifFalse: [self tool handleMouseAt: localEventPosition].
   prevMousePos := localEventPosition
```

Or the user clicks somewhere with button 1:

```
DrGDrawable>>mouseButton1Down: evt localPosition: localPosition
   evt shiftPressed
      ifTrue: [self tool handlePressShiftKey: localPosition ]
      ifFalse: [self tool handlePress: localPosition].
   self showUnderMouseMorph
```

Or when button 2 is released:

```
DrGDrawable>>mouseButton1Up: evt localPosition: localPosition
   evt shiftPressed
      ifTrue: [self tool handleReleaseShiftKey: localPosition]
      ifFalse: [self tool handleRelease: localPosition ].
   (self localBounds containsPoint: localPosition)
      ifFalse: [Cursor normalCursorWithMask activateCursor]
```

The state design pattern works with two hierarchies of classes: the contexts and the states. In DrGeo, a context is the tool currently in use, DrGTool is the root of this hierarchy. A tool can be in various internal states; therefore, the tool dispatches to its internal state the responsibility to handle the user actions:

```
DrGTool>>handleMouseAt: aPoint
   ↑ self state handleMouseAt: aPoint
```

Or

```
DrGTool>>handlePressShiftKey: aPoint
   ↑ self state handlePressShiftKey: aPoint
```

The DrGBuildTool is the tool to construct new items; it comes with a builder as described in Section 7.3 [Builder], page 71:

```
DrGTool subclass: #DrGBuildTool
   instanceVariableNames: 'selectedMorphs builder'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'DrGeo-Presenter-Tool'
```

When a tool is initialized, it is set to a default neutral state:

```
DrGBuildTool>>initialize
   super initialize.
   self switchState: DrGBuildToolState.
   selectedMorphs := OrderedCollection new
```

```
DrGTool>>switchState: aStateClass
   self state: (aStateClass new context: self)
```

The build tool lets the user define a new mathematics item by selecting existing items in a sketch. The relevant message is handlePress::

```
DrGBuildToolState>>handlePress: aPoint
   "Return true if we process something (including additional user choice)"
   | morphs |
   self drawable hideTip.
   self context last: aPoint.
   morphs := self context relevantMorphsNear: aPoint.
   morphs size = 1 ifTrue: [
      self handleChoice: morphs.
      ↑ true].
   (morphs size = 0 and: [self context isWanted: #() ]) ifTrue: [
      self handleChoice: morphs.
      ↑ true].
   "More than one math item under mouse, user must choose one item"
   morphs size >= 2 ifTrue: [
      "Display a pop-up menu to select one item"
      self context chooseMorph: morphs.
      ↑ true].
   ↑ false
```

The DrGBuildTool has only one state. The DrGSelectTool is much more interesting as it comes with 9 different states.

With this tool, the user interacts with the sketch by dragging items, reassigning points, or even cloning items.

```
initialize
   self reset.
   builder := DrGCloneBuilder new
```

The tool is initialized in a neutral state:

```
reset
   super reset.
   start := nil.
   self switchState: DrGSelectToolStateNeutral.
   mathItems := nil.
   builder ifNotNil: [builder reset]
```

It is therefore enlightening to look at the handlers of this state. For example, when hovering a sketch (#handleMouseAt:), the pointer shape is adjusted to a pointing hand to inform the user something can be grabbed:

```
DrGSelectToolStateNeutral>>handleMouseAt: aPoint
   | processSomething |
   (processSomething := super handleMouseAt: aPoint)
      ifTrue: [Cursor webLinkCursor activateCursor]
      ifFalse: [Cursor normalCursorWithMask activateCursor].
   ↑ processSomething
```

More interestingly, when the mouse button is pressed (#handlePress:)on a relevant item, the tool is switched to a grabbed state with class DrGSelectToolStateGrabbed:

```
DrGSelectToolStateNeutral>>handlePress: aPoint
   | morphs griddedPoint|
   self drawable hideTip.
   griddedPoint := self context gridPoint: aPoint.
   morphs := self context relevantMorphsNear: aPoint.
   morphs size = 1
      ifTrue: [
         self context last: griddedPoint.
         self context morph: morphs first.
         self context updateDirtyItemsList.
         self switchState: DrGSelectToolStateGrabbed.
         ↑ true ].
   "More than one math item under mouse"
   morphs size > 1
      ifTrue: [
         self context last: griddedPoint.
         self context chooseMorph: morphs.
         ↑ true ].
   "The user clicked in the background, clear the selection"
   self context reset.
   ↑ false
```

It offers two dedicated handlers. In case the mouse button is released (#handleRelease:) while in grabbed state, the tool switches back to neutral state:

```
DrGSelectToolStateGrabbed>>handleRelease: aPoint
    self switchState: DrGSelectToolStateNeutral.
    self context reset.
    "After move event rehash the the free positionable item"
    self context factory rehash
```

If instead the mouse is moving (#handleMouseAt:), the tool switches to a dragged state with class DrGSelectToolStateDragged:

```
DrGSelectToolStateGrabbed>>handleMouseAt: aPoint
    "The user is moving, switch to dragging state"
    self context
       start: aPoint;
       last: aPoint.
    self context morph isBitmap
       ifTrue: [self switchState: DrGSelectToolStateDraggedBitmap]
       ifFalse: [self drawable dottedLinesToParentsOf: self mathItem.
          self switchState: DrGSelectToolStateDragged].
    ↑ true
```

Understanding how the state of a given tool – or context in the State design pattern terminology – changes is complex to follow. Knowing how the pattern works is fundamental to understand what is going on. This pattern proved to be useful to handle the complexity of a collection of GUI tools, each one with its own way to interact with the UI environment.

Sequence diagrams help to have a global understanding of the flow of states in a given tool.

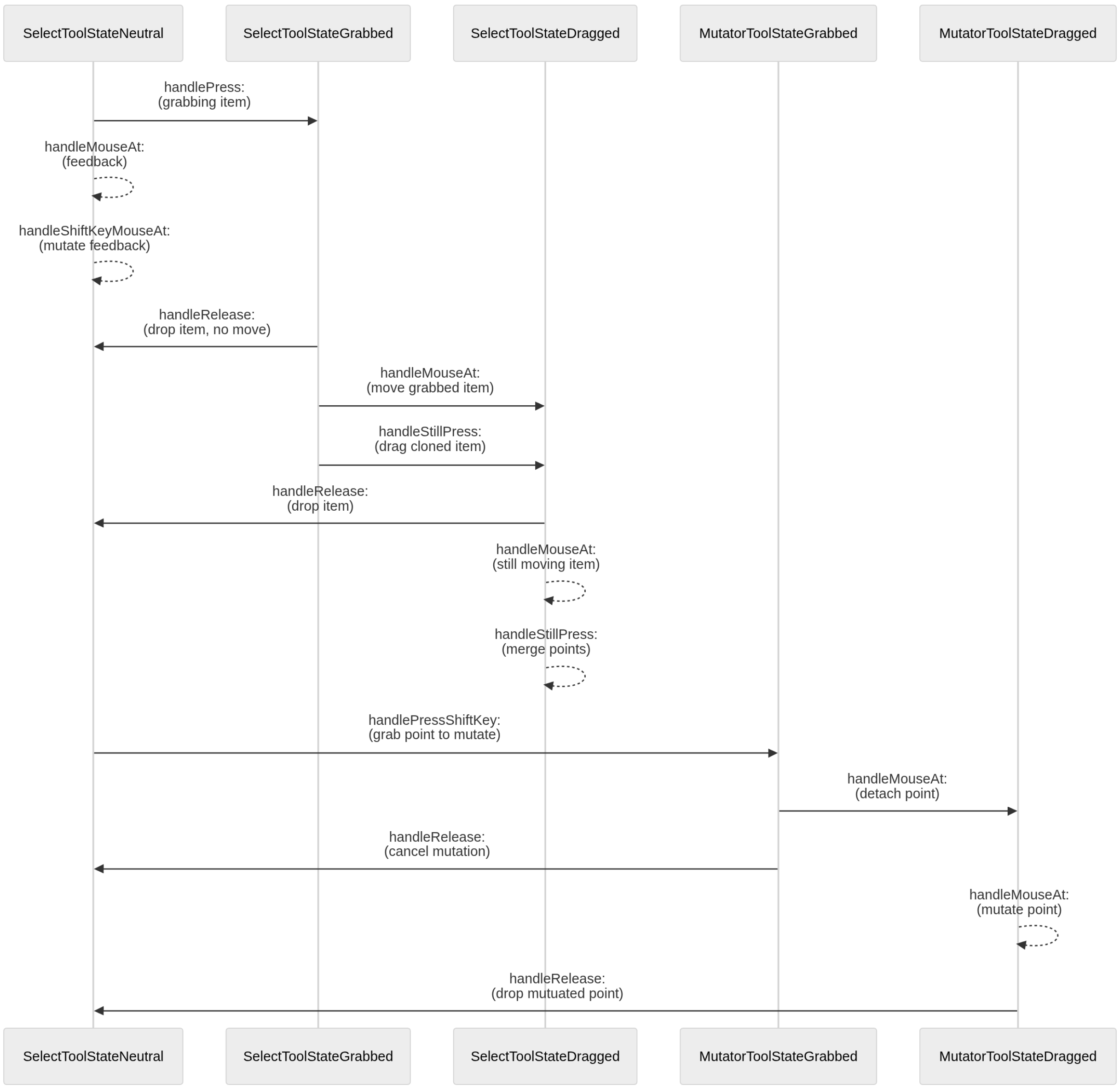

SelectToolStateNeutral
SelectToolStateGrabbed
SelectToolStateDragged
MutatorToolStateGrabbed
MutatorToolStateDragged
handlePress: (grabbing item)
handleMouseAt: (feedback)
handleShiftKeyMouseAt: (mutate feedback)
handleRelease: (drop item, no move)
handleMouseAt: (move grabbed item)
handleStillPress: (drag cloned item)
handleRelease: (drop item)
handleMouseAt: (still moving item)
handleStillPress: (merge points)
handlePressShiftKey: (grab point to mutate)
handleMouseAt: (detach point)
handleRelease: (cancel mutation)
handleMouseAt: (mutate point)
handleRelease: (drop mutuated point)

Figure 7.5: States sequence of the Select tool

7.6 Bridge

The Cuis-Smalltalk graphic system is organized in two hierarchies of classes:

- The MorphicCanvas hierarchy offers 2D drawing services. Notable subclasses are the BitBltCanvas based on the original Smalltalk-80 Bits Block Transfer service, the Vector Graphics service specific to the Cuis-Smalltalk system, and the Hybrid service using simultaneously the two previous services for both efficiency and high rendering quality (HybridCanvas).

- The VectorEngine hierarchy offers the concrete implementation of the graphic rendering. In the sub-hierarchy VectorEngineSmalltalk is the pure Smalltalk implementation for research and debugging purposes. In the sub-hierarchy VectorEngineWithPlugin is the C implementations for optimal speed rendering. Both offer various quality and speed rendering options with specific classes.

The MorphicCanvas hierarchy represents the **Abstraction** part, and the VectorEngine hierarchy is the **Implementation** part of the graphic system. It is an example of the Bridge pattern in the Cuis-Smalltalk system.

The abstraction part represents the public protocol used for drawing the Morphs on the screen. Depending on the type of canvas, different rendering engines are then used.

The AbstractVectorCanvas is the default abstraction used in the Cuis-Smalltalk system. Observe its protocol largely inspired by SVG:

```
Sample01Star>>drawOn: aCanvas
   aCanvas strokeWidth: 12 color: `Color lightOrange` do: [
      aCanvas
         moveTo: `(Point rho: 100 theta: 90 degreesToRadians)`;
         lineTo: `(Point rho: 100 theta: (360/5*2+90) degreesToRadians)`;
         lineTo: `(Point rho: 100 theta: (360/5*4+90) degreesToRadians)`;
         lineTo: `(Point rho: 100 theta: (360/5*6+90) degreesToRadians)`;
         lineTo: `(Point rho: 100 theta: (360/5*8+90) degreesToRadians)`;
         lineTo: `(Point rho: 100 theta: 90 degreesToRadians)` ]
```

DrGeo uses this abstraction to implement its own canvas (DrGSvgCanvas) to render a sketch in a SVG file. It follows the public protocol of AbstractVectorCanvas. The choice was made to make an implementation from the **Abstraction** hierarchy and not the **Implementation** one because the former is used as a parameter in the Morph's drawOn: method.

In See [Sample01Star], page 81, drawing method, the used protocol to implement in DrGSvgCanvas is #strokeWidth:color:do:, #moveTo:, and #lineTo:.

Observe the implementations, in AbstractVectorCanvas:

```
strokeWidth: strokeWidth color: aStrokeColor do: pathCommandsBlock
   "Prepare parameters for drawing stroke."
   self initForPath.
   engine
      strokeWidth: strokeWidth
      color: aStrokeColor
      do: pathCommandsBlock
```

Then in DrGSvgCanvas:

```
strokeWidth: strokeWidth color: strokeColor do: pathCommandsBlock
   self pathStrokeWidth: strokeWidth color: strokeColor fillColor: nil.
   self doPath: pathCommandsBlock ensureClosePath: false
```

And at some point, operations occur in the XML tree:

```
pathStrokeWidth: strokeWidth color: strokeColor strokeDashArray: sda fillColor: fillColor
   pathNode := XMLElement named: #path.
   self
      styleOf: pathNode
      StrokeWidth: strokeWidth
      color: strokeColor
      strokeDashArray: sda
      fillColor: fillColor.
   ↑ pathNode
```

Same goes for lineTo:

```
AbstractVectorCanvas>>lineTo: aPoint
   engine lineTo: aPoint.
   currentPoint := aPoint.
   lastControlPoint := currentPoint
```

```
DrGSvgCanvas>>lineTo: point
   firstPoint ifNil: [↑ self].
   self addPathCommands: 'L ', point printMini.
   currentPoint := point
```

DrGSvgCanvas implements partially the public protocol of AbstractVectorCanvas, restricted to the part used by DrGeo. It weighs only 33 methods, and among them, only 17 are from the canvas abstraction.

7.7 Strategy

To manage the diversity of the operating systems and hosts a Cuis-Smalltalk application is running on, the strategy pattern is handy.

In DrGeo, it is implemented with a hierarchy of DrGPlatform, each one with its own strategy to access resources and preferences in the host: DrGWorkstation to operate on personal computers, DrGDevelopment to operate on developer mode, DrGAndroid and DrGiPad on tablets. The two latter ones are not implemented anymore as DrGeo is not ported anymore to these devices.

Then, in the DrGeoSystem class, it is decided which kind of platform to use, depending on how the system is configured:

```
DrGeoSystem>>beWorkstation
   platform := DrGWorkstation new
```

Or

```
DrGeoSystem>>beDevelopment
   "Be like the Cuis environment for DrGeo development"
   platform := DrGDevelopment new
```

The resources are accessed from the DrGeoSystem public class protocol without knowing which platform is used underneath:

```
DrGeoSystem class>>iconsPath
   ↑ platform iconsPath
```

It depends on the selected platform, only known by DrGeoSystem:

```
DrGPlatform>>iconsPath
   ↑ self resourcesPath / 'icons'
```

```
DrGWorkstation>>iconsPath
   ↑ self resourcesPath / 'graphics' / 'iconsSVG'
```

Appendix A Documents Copyright

Cuis-Smalltalk mascot

The southern mountain cavy (Microcavia australis) is a species of South American rodent in the family Caviidae.
Copyright © Euan Mee

Appendix B The Exercises

Appendix C Solutions to the Exercises

Layout Components (Solutions)

Exercise 2.1

Right after instantiating the boxes, add these two lines:

```
box1 layoutSpec offAxisEdgeWeight: #rowTop.
box3 layoutSpec offAxisEdgeWeight: #rowBottom.
```

box2 is centered by default.

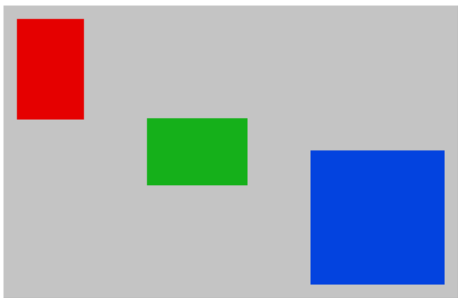


Figure C.1: Top to down

Exercise 2.2

Edit the example as follow:

```
...
spacer1 layoutSpec proportionalWidth: 1.
...
spacer2 layoutSpec proportionalWidth: 3.
...
```

Appendix D The Examples

Appendix E The Figures

Appendix F Memory Game v1

Download MemoryPackage v1 (`https://github.com/DrCuis/DesignGUI/blob/main/misc/MemoryGameV1.pck.st`)

```
'From Cuis7.3 [latest update: #7095] on 10 April 2025 at 10:53:03 am'!
'Description '!
!provides: 'MemoryGameV1' 1 6!
SystemOrganization addCategory: #MemoryGameV1!


!classDefinition: #MemoryCard category: #MemoryGameV1!
PluggableButtonMorph subclass: #MemoryCard
   instanceVariableNames: 'cardColor done'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV1'!
!classDefinition: 'MemoryCard class' category: #MemoryGameV1!
MemoryCard class
   instanceVariableNames: ''!

!classDefinition: #MemoryGameWindow category: #MemoryGameV1!
SystemWindow subclass: #MemoryGameWindow
   instanceVariableNames: 'size cards tupleSize statusBar playground playing'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV1'!
!classDefinition: 'MemoryGameWindow class' category: #MemoryGameV1!
MemoryGameWindow class
   instanceVariableNames: ''!


!MemoryGameWindow commentStamp: '<historical>' prior: 0!
A memory game based on finding identical squares of the same color.!

!MemoryCard methodsFor: 'initialization' stamp: 'hlsf 3/15/2025 15:39:10'!
defaultColor
   ^ Color white! !

!MemoryCard methodsFor: 'initialization' stamp: 'hlsf 3/15/2025 18:38:45'!
initialize
   super initialize.
   done := false! !

!MemoryCard methodsFor: 'accessing' stamp: 'hlsf 3/15/2025 14:48:44'!
cardColor
   "Answer the value of cardColor"

   ^ cardColor! !

!MemoryCard methodsFor: 'accessing' stamp: 'hlsf 3/15/2025 14:48:44'!
cardColor: anObject
   "Set the value of cardColor"

   cardColor := anObject! !

!MemoryCard methodsFor: 'accessing' stamp: 'hlsf 3/15/2025 18:40:42'!
setDone
   done := true.! !

!MemoryCard methodsFor: 'testing' stamp: 'hlsf 3/15/2025 18:38:55'!
isDone
   ^ done! !
```

```
!MemoryCard methodsFor: 'testing' stamp: 'hlsf 3/15/2025 18:21:22'!
isFlipped
   ^ color = cardColor ! !

!MemoryCard methodsFor: 'action' stamp: 'hlsf 3/15/2025 18:21:22'!
flip
   color := self isFlipped ifTrue:  [self defaultColor] ifFalse: [cardColor ].
   self redrawNeeded ! !

!MemoryGameWindow methodsFor: 'testing' stamp: 'hlsf 4/10/2025 10:52:04'!
isGameWon
   ^ (cards elements select: #isDone) size = (size x * size y)! !

!MemoryGameWindow methodsFor: 'testing' stamp: 'hlsf 3/18/2025 23:42:35'!
isPlayed
   ^ playing ! !

!MemoryGameWindow methodsFor: 'testing' stamp: 'hlsf 3/18/2025 23:42:45'!
isStopped
   ^ playing not! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 3/15/2025 18:52:39'!
adoptWidgetsColor: paneColor
" Does nothing, let the buttons have their own colors "! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 4/10/2025 10:51:16'!
doneCards
   ^ cards elements select: #isDone! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 4/10/2025 10:40:18'!
flippedCards
   ^ cards elements select: [:aCard | aCard isDone not and: [aCard isFlipped] ]! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 3/16/2025 17:16:57'!
message: aText
   statusBar contents: aText ;
      redrawNeeded ! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 3/19/2025 16:32:10'!
undoneCards
   ^ cards elements asOrderedCollection
      removeAll: self doneCards;
      yourself.! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/15/2025 18:24:23'!
distributeColors
   | colors |
   colors := OrderedCollection new.
   size x * size y / tupleSize timesRepeat: [colors add: Color random].
   tupleSize - 1 timesRepeat: [colors := colors, colors].
   ^ colors! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/18/2025 23:42:09'!
initialize
   super initialize.
   size := 4 @ 3.
   tupleSize := 2.
   playing := true.
   playground := LayoutMorph newColumn.
   self installToolbar.
   self addMorph: playground.
   self installCards.
   self installStatusBar ! !
```

```
!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/18/2025 23:40:41'!
installCards
   | colors  row |
   playground removeAllMorphs.
   cards := Array2D  newSize: size.
   colors := self distributeColors shuffled .
   1 to: size y do: [:y |
      row := LayoutMorph newRow.
      1 to: size x do: [:x | | card |
         card := MemoryCard model: self action: #flip: actionArgument: x@y.
         card layoutSpec proportionalWidth: 1; proportionalHeight: 1.
         card cardColor: colors removeFirst.
         row addMorph: card.
         cards at: x@y put: card ].
      playground addMorph: row ]! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/18/2025 23:14:00'!
installStatusBar
   statusBar := TextParagraphMorph new
         padding: 2;
         color: Color transparent;
         borderWidth: 1;
         borderColor: self borderColor twiceLighter ;
         setHeightOnContent.
   self addMorph: statusBar layoutSpec: LayoutSizeSpec new useMorphHeight.
   self message: 'Welcome to ', 'Memory Game' bold! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/18/2025 23:30:19'!
installToolbar
   | toolbar button |
   toolbar := LayoutMorph newRow separation: 2.
   button := PluggableButtonMorph model: self action: #startGame ::
      enableSelector: #isStopped;
      icon: Theme current playIcon;
      borderWidth: 2;
      borderColor: Color black;
      setBalloonText: 'Play the game';
      morphExtent: 32 asPoint.
   toolbar addMorph: button.
   button := PluggableButtonMorph model: self action: #stopGame ::
      enableSelector: #isPlayed;
      icon: Theme current stopIcon;
      setBalloonText: 'Stop the game';
      morphExtent: 32 asPoint.
   toolbar addMorph: button.
   self addMorph: toolbar layoutSpec: LayoutSizeSpec new useMorphHeight
! !

!MemoryGameWindow methodsFor: 'updating' stamp: 'hlsf 3/19/2025 16:37:56'!
flip: position
   | flippedCards |
   (cards at: position) flip; lock.

   " Detect if all flipped cards share the same color "
   flippedCards := self flippedCards.
   (flippedCards collect: [:aCard | aCard cardColor]) asSet size = 1 ifFalse: [
      "Give some time for the play to see the color of this card "
      self message: 'Colors do not match!!'.
      self world doOneCycleNow.
      (Delay forSeconds: 1) wait.
      " Color does not match, unflip the flipped card and unlock "
      flippedCards do: [:aCard | aCard flip; unlock].
      ^ self].

   flippedCards size = tupleSize ifTrue: [
      " We found a n-tuple!! "
      self message: 'Great!!' bold, ' You find a ', tupleSize asString, '-tuple!!'.
      flippedCards do: #flash.
      flippedCards do: #setDone.
      self isGameWon ifTrue: [
         self message: 'Congatuluation, you finished the game!!' bold red.
         playing := false] ]! !

!MemoryGameWindow methodsFor: 'updating' stamp: 'hlsf 3/19/2025 16:39:39'!
startGame
   self installCards.
   playing := true.
   self message: 'Starting a new game' bold green! !

!MemoryGameWindow methodsFor: 'updating' stamp: 'hlsf 3/19/2025 16:36:36'!
stopGame
   playing := false.
   self message: 'Game over'.
   self undoneCards do: [:aCard |
      aCard flash; flip.
      self world doOneCycleNow]! !
```

Appendix G Memory Game v2

Download MemoryPackage v2 (`https://github.com/DrCuis/DesignGUI/blob/main/misc/MemoryGameV2.pck.st`)

```
'From Cuis7.3 [latest update: #7095] on 10 April 2025 at 10:56:43 am'!
'Description '!
!provides: 'MemoryGameV2' 1 9!
SystemOrganization addCategory: 'MemoryGameV2'!


!classDefinition: #MemoryCardModel category: 'MemoryGameV2'!
Object subclass: #MemoryCardModel
   instanceVariableNames: 'flipped done color'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV2'!
!classDefinition: 'MemoryCardModel class' category: 'MemoryGameV2'!
MemoryCardModel class
   instanceVariableNames: ''!

!classDefinition: #MemoryGame category: 'MemoryGameV2'!
Object subclass: #MemoryGame
   instanceVariableNames: 'model view playing'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV2'!
!classDefinition: 'MemoryGame class' category: 'MemoryGameV2'!
MemoryGame class
   instanceVariableNames: ''!

!classDefinition: #MemoryGameModel category: 'MemoryGameV2'!
Object subclass: #MemoryGameModel
   instanceVariableNames: 'size tupleSize cards'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV2'!
!classDefinition: 'MemoryGameModel class' category: 'MemoryGameV2'!
MemoryGameModel class
   instanceVariableNames: ''!

!classDefinition: #MemoryGameWindow category: 'MemoryGameV2'!
SystemWindow subclass: #MemoryGameWindow
   instanceVariableNames: 'presenter statusBar playground'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV2'!
!classDefinition: 'MemoryGameWindow class' category: 'MemoryGameV2'!
MemoryGameWindow class
   instanceVariableNames: ''!


!MemoryGame commentStamp: '<historical>' prior: 0!
I am the presenter of the Memory Game. I create the model of the game and I handle the user interaction.!

!MemoryGameWindow commentStamp: '<historical>' prior: 0!
A memory game based on finding identical squares of the same color.!

!MemoryCardModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 13:14:16'!
backColor
   ^ Color white! !

!MemoryCardModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 13:31:11'!
```

```
color
   ^ color! !

!MemoryCardModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 13:31:11'!
color: anObject
   color := anObject! !

!MemoryCardModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 13:30:03'!
setDone
   done := true! !

!MemoryCardModel methodsFor: 'testing' stamp: 'hlsf 3/22/2025 13:30:12'!
isDone
   ^ done! !

!MemoryCardModel methodsFor: 'testing' stamp: 'hlsf 3/22/2025 13:15:39'!
isFlipped
   ^ flipped! !

!MemoryCardModel methodsFor: 'updating' stamp: 'hlsf 3/22/2025 19:40:34'!
flip
   | newColor |
   flipped := flipped not.
   newColor := flipped ifTrue: [color ] ifFalse: [self backColor].
   self triggerEvent: #color with: newColor! !

!MemoryCardModel methodsFor: 'updating' stamp: 'hlsf 3/22/2025 15:53:33'!
flipFlash
   self flip.
   self triggerEvent: #flash! !

!MemoryCardModel methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 13:28:57'!
initialize
   super initialize.
   done := flipped := false! !

!MemoryGame methodsFor: 'callback ui' stamp: 'hlsf 3/29/2025 23:00:43'!
flip: position
   | flippedCards |
   (model cards at: position)
      flip;
      triggerEvent: #lock.
   flippedCards := model flippedCards.
   " Do the flipped cards share the same color? "
   (flippedCards collect: [:aCard | aCard color]) asSet size = 1 ifFalse: [
      " NO "
      " Some delay for the player to see the colors of these flipped cards "
      view message: 'Colors do not match!!'.
      view world doOneCycleNow.
      (Delay forSeconds: 1) wait.
      " Unflip and unlock the flipped cards "
      flippedCards do: [:aCard |
         aCard flip;
            triggerEvent: #flash;
            triggerEvent: #unlock].
      ^ self].

   flippedCards size = model tupleSize ifTrue: [
      " We found a n-tuple!! "
      view message: 'Great!!' bold, ' You find a ', model tupleSize asString, '-tuple!!'.
      flippedCards do: [:aCard | aCard triggerEvent: #flash].
      flippedCards do: #setDone.
      model isGameWon ifTrue: [
         view message: 'Congratuluation, you finished the game!!' bold red.
         playing := false] ]! !
```

```
!MemoryGame methodsFor: 'callback ui' stamp: 'hlsf 3/22/2025 22:28:33'!
startGame
   model installCardModels.
   view installCards.
   view message: 'Starting a new game' bold green.
   view setLabel: 'P L A Y I N G'.
   playing := true.
! !

!MemoryGame methodsFor: 'callback ui' stamp: 'hlsf 3/22/2025 22:27:53'!
stopGame
   playing := false.
   view message: 'Game over'.
   view setLabel: 'G A M E   S T O P P E D'.
   model undoneCards do: [:aCard |
      aCard triggerEvent: #flash; flip.
      view world doOneCycleNow]! !

!MemoryGame methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 22:25:51'!
initialize
   model := MemoryGameModel new.
   view := MemoryGameWindow presenter: self.
   self startGame.
   view openInWorld.! !

!MemoryGame methodsFor: 'testing' stamp: 'hlsf 3/22/2025 15:42:24'!
isPlayed
   ^ playing ! !

!MemoryGame methodsFor: 'testing' stamp: 'hlsf 3/22/2025 15:42:35'!
isStopped
   ^ self isPlayed not! !

!MemoryGame methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 22:07:58'!
model
   ^model! !

!MemoryGameModel methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 22:25:40'!
initialize
   size := 4 @ 3.
   tupleSize := 2! !

!MemoryGameModel methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 15:35:13'!
installCardModels
   | colours |
   cards := Array2D newSize: size.
   colours := self distributeColors.
   1 to: size y do: [:y |
      1 to: size x do: [:x |
         cards
            at: x@y
            put: (MemoryCardModel new color: colours removeFirst) ] ]! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 15:55:08'!
cards
   ^ cards! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 15:09:23'!
distributeColors
   | colors |
   colors := OrderedCollection new.
   size x * size y / tupleSize timesRepeat: [colors add: Color random].
   tupleSize - 1 timesRepeat: [colors := colors, colors].
   ^ colors shuffled! !
```

```
!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 4/10/2025 10:55:03'!
doneCards
   ^ cards elements select: #isDone! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 4/10/2025 10:55:14'!
flippedCards
   ^ cards elements select: [:aCard | aCard isDone not and: [aCard isFlipped] ]! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 15:55:16'!
size
   ^ size! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 19:23:33'!
tupleSize
   ^ tupleSize ! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 15:10:50'!
undoneCards
   ^ cards elements asOrderedCollection
      removeAll: self doneCards;
      yourself.! !

!MemoryGameModel methodsFor: 'testing' stamp: 'hlsf 4/10/2025 10:54:49'!
isGameWon
   ^ (cards elements select: #isDone) size = (size x * size y)! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 3/15/2025 18:52:39'!
adoptWidgetsColor: paneColor
" Does nothing, let the buttons have their own colors "! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 3/16/2025 17:16:57'!
message: aText
   statusBar contents: aText ;
      redrawNeeded ! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 15:38:00'!
presenter: aPresenter
   presenter := aPresenter.
   self model: presenter model! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 22:25:57'!
initialize
   super initialize.
   playground := LayoutMorph newColumn.
   self installToolbar.
   self addMorph: playground.
   self installStatusBar ! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 22:18:57'!
installCards
   | row size |
   playground removeAllMorphs.
   size := model size.
   1 to: size y do: [:y |
      row := LayoutMorph newRow.
      1 to: size x do: [:x | | cardModel cardView |
         cardModel := model cards at: x@y.
         cardView := PluggableButtonMorph model: presenter action: #flip: actionArgument: x@y.
         cardModel
            when: #color send: #color: to: cardView;
            when: #lock send:#lock to: cardView;
            when: #unlock send: #unlock to: cardView;
            when: #flash send: #flash to: cardView.
         cardView layoutSpec proportionalWidth: 1; proportionalHeight: 1.
         cardView color: cardModel backColor.
         row addMorph: cardView].
      playground addMorph: row ]! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/18/2025 23:14:00'!
installStatusBar
   statusBar := TextParagraphMorph new
         padding: 2;
         color: Color transparent;
         borderWidth: 1;
         borderColor: self borderColor twiceLighter ;
         setHeightOnContent.
   self addMorph: statusBar layoutSpec: LayoutSizeSpec new useMorphHeight.
   self message: 'Welcome to ', 'Memory Game' bold! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 15:40:23'!
installToolbar
   | toolbar button |
   toolbar := LayoutMorph newRow separation: 2.
   button := PluggableButtonMorph model: presenter action: #startGame ::
      enableSelector: #isStopped;
      icon: Theme current playIcon;
      borderWidth: 2;
      borderColor: Color black;
      setBalloonText: 'Play the game';
      morphExtent: 32 asPoint.
   toolbar addMorph: button.
   button := PluggableButtonMorph model: presenter action: #stopGame ::
      enableSelector: #isPlayed;
      icon: Theme current stopIcon;
      setBalloonText: 'Stop the game';
      morphExtent: 32 asPoint.
   toolbar addMorph: button.
   self addMorph: toolbar layoutSpec: LayoutSizeSpec new useMorphHeight
! !

!MemoryGameWindow class methodsFor: 'instance creation' stamp: 'hlsf 3/22/2025 15:37:23'!
presenter: aPresenter
   ^ self basicNew
      presenter: aPresenter ;
      initialize ;
      yourself! !
```

Appendix H Memory Game v3

Download MemoryPackage v3 (`https://github.com/DrCuis/DesignGUI/blob/main/misc/MemoryGameV3.pck.st`)

```
'From Cuis7.3 [latest update: #7158] on 24 May 2025 at 4:26:24 pm'!
'Description '!
!provides: 'MemoryGameV3' 1 3!
SystemOrganization addCategory: 'MemoryGameV3'!


!classDefinition: #Command category: 'MemoryGameV3'!
Object subclass: #Command
   instanceVariableNames: 'presenter'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV3'!
!classDefinition: 'Command class' category: 'MemoryGameV3'!
Command class
   instanceVariableNames: ''!

!classDefinition: #PlayCardCommand category: 'MemoryGameV3'!
Command subclass: #PlayCardCommand
   instanceVariableNames: 'status position'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV3'!
!classDefinition: 'PlayCardCommand class' category: 'MemoryGameV3'!
PlayCardCommand class
   instanceVariableNames: ''!

!classDefinition: #CommandManager category: 'MemoryGameV3'!
Object subclass: #CommandManager
   instanceVariableNames: 'stack presenter'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV3'!
!classDefinition: 'CommandManager class' category: 'MemoryGameV3'!
CommandManager class
   instanceVariableNames: ''!

!classDefinition: #MemoryCardModel category: 'MemoryGameV3'!
Object subclass: #MemoryCardModel
   instanceVariableNames: 'flipped done color'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV3'!
!classDefinition: 'MemoryCardModel class' category: 'MemoryGameV3'!
MemoryCardModel class
   instanceVariableNames: ''!

!classDefinition: #MemoryGame category: 'MemoryGameV3'!
Object subclass: #MemoryGame
   instanceVariableNames: 'model view playing cmdManager'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV3'!
!classDefinition: 'MemoryGame class' category: 'MemoryGameV3'!
MemoryGame class
   instanceVariableNames: ''!

!classDefinition: #MemoryGameModel category: 'MemoryGameV3'!
Object subclass: #MemoryGameModel
```

```
   instanceVariableNames: 'size tupleSize cards'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV3'!
!classDefinition: 'MemoryGameModel class' category: 'MemoryGameV3'!
MemoryGameModel class
   instanceVariableNames: ''!

!classDefinition: #CommandStack category: 'MemoryGameV3'!
ReadWriteStream subclass: #CommandStack
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV3'!
!classDefinition: 'CommandStack class' category: 'MemoryGameV3'!
CommandStack class
   instanceVariableNames: ''!

!classDefinition: #MemoryGameWindow category: 'MemoryGameV3'!
SystemWindow subclass: #MemoryGameWindow
   instanceVariableNames: 'presenter statusBar playground'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MemoryGameV3'!
!classDefinition: 'MemoryGameWindow class' category: 'MemoryGameV3'!
MemoryGameWindow class
   instanceVariableNames: ''!


!Command commentStamp: '<historical>' prior: 0!
An abstract class to represent commands of user actions in the game.
presenter is the game presenter!

!CommandManager commentStamp: '<historical>' prior: 0!
A manager for user commands. At user actions, I create specific instances of command recorded in a stack.!

!MemoryGame commentStamp: '<historical>' prior: 0!
I am the presenter of the Memory Game. I create the model of the game and I handle the user interaction.!

!CommandStack commentStamp: '<historical>' prior: 0!
I am stack of command to execute or to unexecute user action in a Document!

!MemoryGameWindow commentStamp: '<historical>' prior: 0!
A memory game based on finding identical squares of the same color.!

!Command methodsFor: 'command' stamp: 'hlsf 9/10/2024 21:05:57'!
execute
   self subclassResponsibility ! !

!Command methodsFor: 'command' stamp: 'hlsf 9/10/2024 21:06:19'!
unexecute
   self subclassResponsibility ! !

!Command methodsFor: 'accessing' stamp: 'hlsf 5/23/2025 17:18:48'!
presenter: aPresenter
   presenter := aPresenter ! !

!Command methodsFor: 'initialize-release' stamp: 'hlsf 9/10/2024 21:06:12'!
release
"Let my child do some clean up"! !

!Command class methodsFor: 'instance creation' stamp: 'hlsf 5/23/2025 19:14:23'!
for: aPresenter
   ^ self basicNew
      presenter: aPresenter ;
      initialize ! !

!PlayCardCommand methodsFor: 'updating' stamp: 'hlsf 5/23/2025 21:50:41'!
backupModels
   | size |
   size := presenter model cards size.
   1 to: size y do: [:y |
         1 to: size x do: [:x | | card |
            card := presenter model cards at: x@y.
            status at: x@y put: (Array with: card isFlipped  with: card isDone) ]]! !

!PlayCardCommand methodsFor: 'initialization' stamp: 'hlsf 5/23/2025 21:51:00'!
initialize
   status := Array2D newSize: presenter model cards size.
   ! !

!PlayCardCommand methodsFor: 'accessing' stamp: 'hlsf 5/23/2025 22:11:01'!
position: aPoint
   position := aPoint! !

!PlayCardCommand methodsFor: 'command' stamp: 'hlsf 5/23/2025 22:11:47'!
execute
   self backupModels.
   presenter flip: position! !

!PlayCardCommand methodsFor: 'command' stamp: 'hlsf 5/23/2025 22:26:00'!
unexecute
" Restore the status of the card models "
   | size |
   size := status size.
   1 to: size y do: [:y |
         1 to: size x do: [:x | | cardStatus card |
            card := presenter model cards at: x@y.
            cardStatus := status at: x@y.
            card
               flip: cardStatus first;
               done: cardStatus second ] ]! !

!CommandManager methodsFor: 'initialize-release' stamp: 'hlsf 5/24/2025 10:00:33'!
initialize
   stack := CommandStack new! !

!CommandManager methodsFor: 'initialize-release' stamp: 'hlsf 5/24/2025 10:00:33'!
release
   stack contents do: [:c | c release].
   stack reset! !

!CommandManager methodsFor: 'commands' stamp: 'hlsf 5/24/2025 10:13:46'!
playCard: position
   | command |
   command := stack nextPut: (PlayCardCommand for: presenter).
   command position: position.
   ^ command execute! !

!CommandManager methodsFor: 'commands' stamp: 'hlsf 5/24/2025 10:00:33'!
redo
   | command |
   command := stack next.
   ^ command
      ifNotNil:    [
         command execute.
         true]
      ifNil: [false]! !

!CommandManager methodsFor: 'commands' stamp: 'hlsf 5/24/2025 10:00:33'!
```

```
undo
   | command |
   command := stack previous.
   ^ command
      ifNotNil: [
         command unexecute.
         true]
      ifNil: [false]! !

!CommandManager methodsFor: 'accessing' stamp: 'hlsf 5/23/2025 17:13:23'!
presenter: aPresenter
   presenter := aPresenter ! !

!CommandManager methodsFor: 'testing' stamp: 'hlsf 5/24/2025 10:02:15'!
canRedo
   ^ stack atEnd not! !

!CommandManager methodsFor: 'testing' stamp: 'hlsf 5/24/2025 10:02:06'!
canUndo
   ^ stack atStart not! !

!MemoryCardModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 13:14:16'!
backColor
   ^ Color white! !

!MemoryCardModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 13:31:11'!
color
   ^ color! !

!MemoryCardModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 13:31:11'!
color: anObject
   color := anObject! !

!MemoryCardModel methodsFor: 'accessing' stamp: 'hlsf 5/23/2025 22:22:01'!
done: boolean
   done = boolean ifTrue: [^ self].
   done := boolean.
   self triggerEvent: (done ifTrue: [#lock] ifFalse: [#unlock])! !

!MemoryCardModel methodsFor: 'accessing' stamp: 'hlsf 5/23/2025 22:18:00'!
flip: boolean
" Set my flip state and trigger a color event for my view accordingly to my flip state "
   | newColor |
   flipped = boolean ifTrue: [^ self].
   flipped := boolean.
   newColor := flipped ifTrue: [color ] ifFalse: [self backColor].
   self triggerEvent: #color with: newColor! !

!MemoryCardModel methodsFor: 'testing' stamp: 'hlsf 3/22/2025 13:30:12'!
isDone
   ^ done! !

!MemoryCardModel methodsFor: 'testing' stamp: 'hlsf 3/22/2025 13:15:39'!
isFlipped
   ^ flipped! !

!MemoryCardModel methodsFor: 'updating' stamp: 'hlsf 5/23/2025 21:56:48'!
flip
   " Reverse my flip state "
   self flip: flipped not! !

!MemoryCardModel methodsFor: 'updating' stamp: 'hlsf 5/23/2025 21:57:23'!
flipFlash
   " Flip & trigger a flash event for my view "
   self flip.
   self triggerEvent: #flash! !

!MemoryCardModel methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 13:28:57'!
initialize
   super initialize.
   done := flipped := false! !

!MemoryGame methodsFor: 'callback ui' stamp: 'hlsf 5/23/2025 21:53:20'!
flip: position
   | flippedCards |
   (model cards at: position)
      flip;
      triggerEvent: #lock.
   flippedCards := model flippedCards.
   " Do the flipped cards share the same color? "
   (flippedCards collect: [:aCard | aCard color]) asSet size = 1 ifFalse: [
      " NO "
      " Some delay for the player to see the colors of these flipped cards "
      view message: 'Colors do not match!!'.
      view world doOneCycleNow.
      (Delay forSeconds: 1) wait.
      " Unflip and unlock the flipped cards "
      flippedCards do: [:aCard |
         aCard flip;
            triggerEvent: #flash;
            triggerEvent: #unlock].
      ^ self].

   flippedCards size = model tupleSize ifTrue: [
      " We found a n-tuple!! "
      view message: 'Great!!' bold, ' You find a ', model tupleSize asString, '-tuple!!'.
      flippedCards do: [:aCard |
         aCard triggerEvent: #flash;
         done: true].
      model isGameWon ifTrue: [
         view message: 'Congratuluation, you finished the game!!' bold red.
         playing := false] ]! !

!MemoryGame methodsFor: 'callback ui' stamp: 'hlsf 3/22/2025 22:28:33'!
startGame
   model installCardModels.
   view installCards.
   view message: 'Starting a new game' bold green.
   view setLabel: 'P L A Y I N G'.
   playing := true.
! !

!MemoryGame methodsFor: 'callback ui' stamp: 'hlsf 5/24/2025 16:24:36'!
stopGame
   playing := false.
   cmdManager release.
   view message: 'Game over'.
   view setLabel: 'G A M E   S T O P P E D'.
   model undoneCards do: [:aCard |
      aCard triggerEvent: #flash; flip.
      view world doOneCycleNow]! !

!MemoryGame methodsFor: 'initialization' stamp: 'hlsf 5/24/2025 10:13:20'!
initialize
   model := MemoryGameModel new.
   cmdManager := CommandManager new :: presenter: self.
   view := MemoryGameWindow presenter: self.
   self startGame.
   view openInWorld.! !
```

```
!MemoryGame methodsFor: 'testing' stamp: 'hlsf 5/24/2025 10:02:37'!
canRedo
   ^ cmdManager canRedo ! !

!MemoryGame methodsFor: 'testing' stamp: 'hlsf 5/24/2025 10:02:28'!
canUndo
   ^ cmdManager canUndo ! !

!MemoryGame methodsFor: 'testing' stamp: 'hlsf 3/22/2025 15:42:24'!
isPlayed
   ^ playing ! !

!MemoryGame methodsFor: 'testing' stamp: 'hlsf 3/22/2025 15:42:35'!
isStopped
   ^ self isPlayed not! !

!MemoryGame methodsFor: 'accessing' stamp: 'hlsf 5/24/2025 10:07:05'!
cmdManager
   ^ cmdManager ! !

!MemoryGame methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 22:07:58'!
model
   ^model! !

!MemoryGameModel methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 22:25:40'!
initialize
   size := 4 @ 3.
   tupleSize := 2! !

!MemoryGameModel methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 15:35:13'!
installCardModels
   | colours |
   cards := Array2D newSize: size.
   colours := self distributeColors.
   1 to: size y do: [:y |
      1 to: size x do: [:x |
         cards
            at: x@y
            put: (MemoryCardModel new color: colours removeFirst) ] ]! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 15:55:08'!
cards
   ^ cards! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 15:09:23'!
distributeColors
   | colors |
   colors := OrderedCollection new.
   size x * size y / tupleSize timesRepeat: [colors add: Color random].
   tupleSize - 1 timesRepeat: [colors := colors, colors].
   ^ colors shuffled! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 4/10/2025 10:55:03'!
doneCards
   ^ cards elements select: #isDone! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 4/10/2025 10:55:14'!
flippedCards
   ^ cards elements select: [:aCard | aCard isDone not and: [aCard isFlipped] ]! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 15:55:16'!
size
   ^ size! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 19:23:33'!
```

```
tupleSize
   ^ tupleSize ! !

!MemoryGameModel methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 15:10:50'!
undoneCards
   ^ cards elements asOrderedCollection
      removeAll: self doneCards;
      yourself.! !

!MemoryGameModel methodsFor: 'testing' stamp: 'hlsf 4/10/2025 10:54:49'!
isGameWon
   ^ (cards elements select: #isDone) size = (size x * size y)! !

!CommandStack methodsFor: 'private' stamp: 'hlsf 9/10/2024 21:00:31'!
truncate
   |oldReadLimit|
   oldReadLimit := readLimit.
   readLimit := position.
   oldReadLimit > readLimit ifTrue:
         [readLimit to: oldReadLimit do:
            [:index| collection at: index + 1 put: nil]]! !

!CommandStack methodsFor: 'accessing' stamp: 'hlsf 9/10/2024 21:00:20'!
nextPut: aCommand
   super nextPut: aCommand.
   self truncate.
   ^ aCommand ! !

!CommandStack methodsFor: 'accessing' stamp: 'hlsf 9/10/2024 21:00:26'!
previous
   self position = 0 ifTrue: [^nil].
   self position: self position - 1.
   ^self peek.! !

!CommandStack methodsFor: 'as yet unclassified' stamp: 'hlsf 10/26/2024 14:58:31'!
reset
   super reset.
   self truncate! !

!CommandStack class methodsFor: 'instance creation' stamp: 'hlsf 9/10/2024 21:01:01'!
new
   ^self on: Array new! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 3/15/2025 18:52:39'!
adoptWidgetsColor: paneColor
" Does nothing, let the buttons have their own colors "! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 3/16/2025 17:16:57'!
message: aText
   statusBar contents: aText ;
      redrawNeeded ! !

!MemoryGameWindow methodsFor: 'accessing' stamp: 'hlsf 3/22/2025 15:38:00'!
presenter: aPresenter
   presenter := aPresenter.
   self model: presenter model! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/22/2025 22:25:57'!
initialize
   super initialize.
   playground := LayoutMorph newColumn.
   self installToolbar.
   self addMorph: playground.
   self installStatusBar ! !
```

```
!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 5/24/2025 10:08:12'!
installCards
   | row size |
   playground removeAllMorphs.
   size := model size.
   1 to: size y do: [:y |
      row := LayoutMorph newRow.
      1 to: size x do: [:x | | cardModel cardView |
         cardModel := model cards at: x@y.
         cardView := PluggableButtonMorph
            model: presenter cmdManager action: #playCard: actionArgument: x@y.
         cardModel
            when: #color send: #color: to: cardView;
            when: #lock send:#lock to: cardView;
            when: #unlock send: #unlock to: cardView;
            when: #flash send: #flash to: cardView.
         cardView layoutSpec proportionalWidth: 1; proportionalHeight: 1.
         cardView color: cardModel backColor.
         row addMorph: cardView].
      playground addMorph: row ]! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 3/18/2025 23:14:00'!
installStatusBar
   statusBar := TextParagraphMorph new
         padding: 2;
         color: Color transparent;
         borderWidth: 1;
         borderColor: self borderColor twiceLighter ;
         setHeightOnContent.
   self addMorph: statusBar layoutSpec: LayoutSizeSpec new useMorphHeight.
   self message: 'Welcome to ', 'Memory Game' bold! !

!MemoryGameWindow methodsFor: 'initialization' stamp: 'hlsf 5/24/2025 10:32:50'!
installToolbar
   | toolbar button |
   toolbar := LayoutMorph newRow separation: 2.
   button := PluggableButtonMorph model: presenter action: #startGame ::
      enableSelector: #isStopped;
      icon: Theme current playIcon;
      setBalloonText: 'Play the game';
      morphExtent: 32 asPoint.
   toolbar addMorph: button.
   button := PluggableButtonMorph model: presenter action: #stopGame ::
      enableSelector: #isPlayed;
      icon: Theme current stopIcon;
      setBalloonText: 'Stop the game';
      morphExtent: 32 asPoint.
   toolbar addMorph: button.
   button := PluggableButtonMorph model: presenter cmdManager action: #undo ::
      enableSelector: #canUndo;
      icon: Theme current undoIcon ;
      setBalloonText: 'Undo the last play';
      morphExtent: 32 asPoint.
   toolbar addMorph: button.
   button := PluggableButtonMorph model: presenter cmdManager action: #redo ::
      enableSelector: #canRedo;
      icon: Theme current redoIcon ;
      setBalloonText: 'Redo the last play';
      morphExtent: 32 asPoint.
   toolbar addMorph: button.
   self addMorph: toolbar layoutSpec: LayoutSizeSpec new useMorphHeight
! !

!MemoryGameWindow class methodsFor: 'instance creation' stamp: 'hlsf 3/22/2025 15:37:23'!
presenter: aPresenter
```

```
^ self basicNew
   presenter: aPresenter ;
   initialize ;
   yourself! !
```

Appendix I Conceptual index