Little Smalltalk - MacGUI

Dr Sam Sandqvist

January 2023

The Little Smalltalk implementation originally created by Dr Budd in the 80s has always been text-based, with some attempts to create a modern GUI environment in the past. There is also a WEB GUI, which provides a nice interface for web applications.

The present implementation provides this missing piece for the Mac, and finally gives us the opportunity to create application with a *native*, modern look in Smalltalk for this system.

Note that this is still work in progress, as is this sketchy document highlighting the new UI and its usage.

Philosophy

The system is of course based on Apple-provided MacOS UI elements, which are originally written in Objective-C (an object-oriented superset of C), and now slowly being enhanced by Apple with capabilities written in their Swift language.

Swift is nice, modern, and performs well. But it is also huge, and results in quite large applications. Objective-C is, like C, very picky but provides excellent performance, and small executable size (since all libraries are part of the system).

Smalltalk, being a dynamic, highly productive system, provides very small size, but not high performance. However, in GUI applications, this normally does not matter that much.

The Smalltalk implementation is in reality written in a mixture of C and Objectice-C, and provides a quite thin wrapper on the Apple-provided frameworks, with some notable exceptions as will be highlighted below.

The system is used in a very simple manner, as the following example shows.

```
|w ....|                    " define variables
w := Window newWindow.  " this initiliases the application behind the scenes "
...                        " create some UI elements and their actions/notifications "
w addPane: ...             " add them to the window (or panes)
MacApp run.                " and off we go! "
```

A central part of the implementation is event and action handling. This is done consistently with **blocks**. For instance

```
aButton mouseDown: ['Button says that it is impressed!' printNl].
```

There are essentially two kinds of callback mechanisms: *actions* and *notifications*. Actions are almost universally used; e.g., all controls and mouse events use actions (like the example above), as do Menus. Notifications are currently used very rarely, as they are more system-wide and we need to get the sender as well as the name of the notification to do anything. The `:sender` and `:name` arguments should always be specified for notifications.

So, we always provide two block temporaries: `:sender` and `:name`; they provide a convenient way of discerning who sent the message and executed the block, and what was the notification name. Note that we do not trap all notifications, just the most important ones. Also note that the `sender` is the object that caused the action.

Also note that very UI element has a hidden `pointer` that references the object internally. **Never** modify the pointer of an element, or the system will certainly crash when trying to use it!

Also note that `action: aBlock` messages does not have a temporary argument, the sender is of course the object that the `action:` or similar message was sent to, and the action name (e.g., `mouseDown:`) provides the name of the action.

The menu system also uses blocks to indicate what should be executed on menu selections.

The menu system

Since the menu system is quite complicated, but virtually every application should use one, it is described in some more detail here.

In general, the menu is a menu of `MenuItems`, added to the menu bar (the main menu). A `MenuItem` is created using

```
anItem := MenuItem title: aString key: aKey action: aBlock
```

where the title will be shown, followed by the shortcut key (automatically prefixed with COMMAND). On selection, the block will be executed. If you don't want a shortcut key, enter an empty string.

On top of the screen we have the menu bar, in common with all MacOS applications. The leftmost menu is almost standard, and may be created as sketched below for an app `MyApp`, using the supplied `addMenu: aString items: aList` method.

```
|mb m1 ...|
mb := MacApp menu.

m1 := List new.

m1 addLast: (MenuItem title: 'About MyApp' key: '' action: [
    Alert title: 'About MyApp' message: ('MyApp V1.0', (10 asChar asString), 'Copyright ©
]).
m1 addLast: (MenuItem separator).
m1 addLast: (MenuItem title: 'Preferences...' key: ',' action: [myApp preferences]).
m1 addLast: (MenuItem separator).
m1 addLast: (MenuItem title: 'Quit MyApp' key: 'q' action: [MacApp close]).

mb addMenu: 'MyApp' items: m1.
```

Note that the 'MyApp' string above will be the name of the (leftmost) menu, if you have created a GUI application. (if you use the command-line application with GUI interface, the app name will not change from the default `lst`). Typically, the next menu would be 'File', 'Edit', and so on, with the last 'Help'. If you use these names, some of the standard items for these menus will be added automatically.

Also note that we use the `Alert title: aString message: aString` to show an alert box.

Context menus may be added in almost the same way. Create as before, but add to the `Pane`, or its derivatives:

```
myPane := Pane new; ....

m := Menu title: 'Whatever'.
m item: (MenuItem ...).
m item: (MenuItem ...)
...
myPane menu: myMenu.
```

This menu may then be accessed with a right-click (or CTRL-click) on the mouse. The previously mentioned `addMenu:items:` method constructs the menu using these commands.

ScrollPane

The concept of a scrollable region is used quite often as a UI element in applications. In MacOS this is implemented using a `ScrollPane`. Unfortunately, its use is not quite intuitive.

Generally, by executing a command similar to

```
sc := ScrollPane new; x: 10 y: 10 height: 400 width 200.
```

one defines a region on the window that may contain a pane the contents of which may be scrolled. By itself, it doesn't do anything.

The normal use is to add another pane to the scrollpane; this pane is large, e.g., an outline or table, which may grow as items are shown/expanded/collapsed or added. So, one way would be to say

```
w := Window new.
...
op := OutlinePane new; x: 0 y: 0 width: 200 height: 1000.
...
sc addPane: op.
w addPane: sc.
```

This would make the `OutlinePane` scrollable inside the defined region, inside the window `w`. When you scroll you may obtain the visible portion using the `scrollRectangle` message. It provides an `Array` with four elements: x, y, width, and height.

Class hierarchy

At present, there are the following classes.

- `Application` provided by the LittleSmalltalk system
- `MacApp` base system class (subclass of `Application`, provides constants and methods supporting the system. Never instantiated, as there can never be more than one application.
- `Window` base class for all UI elements. Must be instantiated once, but one may create several windows (e.g., one for Preferences is quite common).
    - `Pane` represents a rectangular area in your `Window`. It has some properties (like position and size, backgound colour, border width and colour) but not really anything else. However, it may host other `Panes`, and all other UI elements.
        - `Control` abstract class, providing common functionality for all UI elements below
            - `Button` various types of buttons
            - `TextBox` multi-line text entry
            - `TextField` one-line text entry
                - `Label` one-line label (non-editable field, without border)
            - `ColorPanel` provide a way of choosing colours.
        - `ScrollPane` provide a scrollable area. To be filled in with another Pane.
        - `ImagePane` to hold images.
        - `TablePane` to hold tabular information. (TBC)
        - `OutlinePane` to hold hierarchical and tabular information. (TBC)

- `Point` provides the standard way of using locatons, namely `x@y` and sometimes `width@height`.
- `Color` provides a way of creating colours from red, green, blue and alpha values, as well as some standard pre-defined colours.
- `Pointer` provides access to the mouse position at all times.
- `Font` create fonts for the labels and other control elements.
- `Menu` create menus both application-wide and context.
- `MenuItem` create single items for menus.
- `Alert` provide a modal alert message.
- `Image` basically a ByteArray used for (raw) image data.
- `Time` provide access to system time.
- `Timer` provides capability to execute something after a set time.

Most of the functionality is in the methods.

Let's look at each Class and its public methods.

Methods

Note that we only provide documentation for publicly useful methods. There are many more that should not be used. Furthermore, the standard `new` method is not mentioned; should be used unless otherwise indicated (e.g., `Point`). Also, sometimes a method is only mentioned higher in the hierarchy but of course applicable lower as well.

1. `MacApp`. All class methods.
    - `mainWindow` provides an opaque pointer to the application's main window. Do not change this value. `nil` if the system has not been initialised.
    - `homeDirectory` provides system information
    - `documentDirectory` provides system information
    - `executableDirectory` provides system information
    - `resourceDirectory` provides system information
    - `openFile` if the user clicked a file that caused the application to start via a file extension association, the file name will be stored in the class variable `openFile`. Will be `nil` otherwise.
    - `start` start the application, set up `onStartup:` and `onClose:` blocks
    - `run` start the application, without setting up possible `onStartup:` and `onClose:` blocks.
    - `onStart: aBlock` provide block to be executed when system initialised and ready
    - `onClose: aBlock` provide block to be executed before the application shuts down.
    - `close` halt the application. Note that this is not the same as `close` for a window your application has created. Also, by default a `MacApp` quits when its last window is closed.
    - `notification: aBlock` set a block to be executed for each action on the app. Note that this is provided for lower level control. See discussion about actions and notifications.
    - `menu` provides the application-wide menu.
    - `openPanel: aType multi: aBoolean dir: aBoolean` opens a file open panel, allowing

opening of files of type `aType`. Allow multiple files to be opened with `multi`, and directories with `dir`. If both `multi` and `dir` are specified, all files in the chosen directory will be returned. The `openPanel` message will always return an `Array` with the chosen information; none (zero size) if cancelled.
- `openPanel: aType` short form of the above, with single file and no directory.
- `savePanel: aString` opens a file save dialogue, and returns either the saved file name, or empty if cancelled. `aString` will be offered as default, if specified.
- `copyFile: aFilename to: anotherName` copy files quickly.

2. `Window`. All instance methods

- `newWindow` create a new window. **Always use this to create windows, not the plain `new` as that will not initialise things properly.**
- `show` show a newly created window. Do not use for mainWindow; that is shown automatically.
- `close` close a window. Do not use for mainWindow; halt the application with `MacApp close` instead.
- `onResize: aBlock` block to be executed when the user resizes the mainWindow.
- `x: xPos y: yPos width: aWidth height: aHeight` set window dimensions and location. Note that windows and panes also respond to `x`, `y`, `width`, and `height`, as well as the corresponding setters (`x:` etc.).
- `title: aString` set window title.
- `center` centre the window on the screen.
- `addPane: aPane` add a UI element (pane or control) to the window. Nothing shows unless added to a pane (that is added to a window) or directly to the window

3. `Pane`. Most of these methods apply to the descendant elements as well.

- `x: xPos y: yPos width: aWidth height: aHeight` set pane dimensions and location.
- `origin` obtain the x and y coordinate `Point` of the location of the pane (or element)
- `origin: aPoint` move the pane or element to the new position
- `size` obtain the width and height of the pane or element as a `Point`.
- `size: aPoint` resize the pane or element.
- `mouseXXX: aBlock modifier: aKey` execute block for the action in the pane. The action is defined by the XXX as follows:

    - XXX = Down, mouse clicked down
    - XXX = Up, mouse let go
    - XXX = Dragged, mouse dragged

- Modifiers are listed below; may be added together for composite key presses
    - 0 = normal
    - 1 = SHIFT pressed
    - 2 = CTRL pressed
    - 4 = ALT / OPTION pressed
    - 8 = COMMAND pressed

- `mouseXXX: aBlock` execute the block for the action in the pane, with modifier (see above) as 0 (no additional key pressed)
- `rightMouseXXX: aBlock modifier: aKey` execute the block for the action in the pane, but with the right hand side of the mouse.
- `rightMouseXXX: aBlock` execute the block for the action in the pane, but with the right hand side of the mouse, and no modifier.
- `doubleClick: aBlock` execute the block for double left clicks, with or without modifier as above.
- `keyDown: aKeyCode action: aBlock modifier: aKey` executed on keypresses on the pane, in the same manner as mouse clicks, and same modifier values as above. Nb: Care must be taken that the Pane has the focus and can receive keystrokes. Also, the `keyCode` is Apple-internal (and keyboard-specific) *(TBC: automatic conversion of characters to key codes)*
- `keyUp: aKeyCode action: aBlock modifier: aKey` likewise, for key let go.
- `keyDown: aKeyCode action: aBlock` executed on keypresses on the pane, in the same manner as mouse clicks; no modifier.
- `keyUp: aKeyCode action: aBlock` likewise, for key let go and no modifier.
- `borderWidth: aWidth` set the border width a given thickness (pixels). `border:` may also be used.
- `backgroundColor: aColour` set the colour of the pane. `color:` may also be used.
- `borderColor: aColour` set the colour of the border.
- `show` and `hide` temporarily hide and show the pane.
- `isHidden` return `true` or `false` depending on the visibility status of the pane.
- `remove` permanently remove the pane; recreate if needed again.
- `focus` move mouse and entry focus to the pane.
- `menu: aMenu` add a context menu to the pane. See discussion about the menu system above.
- `corners: anInteger` round the corners of the pane with the radius of `anInteger` pixels.
- `shadow: anInteger opacity: aPercent` provide the pane with a shadow of size `anInteger` and given opacity (in percent, 0..100).

4. `ScrollPane` used to hold other panes; see discussion above.

    - `scrollRectangle` the frame of the visible portion of the `ScrollPane`. An `Array`, with x, y, width, and height.

5. `ImagePane` used to show images

    - `image: anImage size: aSize scale: aScaleMode` set the image `anImage` (an instance of `Image`) to be shown on the `ImagePane`. The size should be obtained using `size` method on the instance of `File` where the image originated. The scale modes are the following:

        - 0 = center the image in the pane
        - 1 = resize the image so that it fits in the pane (possibly distorting its aspect)
        - 2 = resize the image so that it fits in the pane keeping its aspect (and showing the whole image)

        - 3 = resize the image so that it fills the image keeping its aspect (possibly leaving parts outside; centered on the axis it exceeds)

    - `image: anImage` same as above, but the size is taken from the image and the scale mode is set to 2.

6. `Image` a subclass of `ByteArray`

    - `size` returns the size of the image.

7. `Pointer`. Only method is the following, applied to the class

    - `location` provides the mouse pointer location as a `Point`, that is `x@y`.

8. `Color`. Special class method to create a colour

    - `newRed: r green: g blue: b alpha: a` where all values are integers 0..255.
    - `red`, `green`, `blue`, `white`, `black`, `grey`, `orange` short methods for creating common colours (to be expanded).

9. `ColorPanel` choose colours from the system colour dialogue panel. Note that this is common to all applications, and will show the last chosen colour unless explicitly changed.

    - `color` obtain the chosen colour.
    - `color: aColor` set the colour
    - `action: aBlock` execute the block on colour choice
    - `show` and `hide` shows and hides the panel. **Note that by default the `ColorPanel` is not visible when created. Use `show` to make it visible.** Typical use is to have a separate button or pane that when clicked shows the panel, and then an action block that obtains it and hides the panel.

10. `Button` Note that some methods are not applicable to all types of Buttons.

    - `title: aString` display string on button.
    - `style: aStyle` set style. Styles are (nb. this is from Apple; some styles are not obvious):

        - 1 = rounded
        - 2 = regular square
        - 5 = disclosure
        - 6 = shadowless square
        - 7 = circular
        - 8 = textured square
        - 9 = help
        - 10 = small square
        - 11 = textured rounded
        - 12 = rounded rectangular
        - 13 = recessed

        - 14 = rounded disclosure
        - 15 = inline
    - `type: aType` set type. Types are (nb. this is from Apple; some types are not obvious):
        - 0 = momentaryLight
        - 1 = pushOnOff
        - 2 = toggle
        - 3 = switch (checkbox)
        - 4 = radio
        - 5 = momentaryChange
        - 6 = onOff
        - 7 = momentaryPushIn
        - 8 = accelerator
        - 9 = multilevel accelerator
    - `state` activated (1) or deactivated (0).
    - `state: aState` set state; 0 off/not activated, non-zero on/activated.
    - `key: aString` typing this will activate the button. Note that there are two common special cases: `13 asChar asString` sets the ENTER key, and makes the button the default button (normally indicated by being blue), and `27 asChar asString` sets the ESCAPE key, denoting Cancel.
    - `setAsDefault` will set the button to the default (see above)
    - `setAsCancel` will set the button to be cancel (see above)
    - `action: aBlock` set the block to be executed when the key is pressed (activated)
11. `TextField` and `Label`
    - `string: aString` sets the default for the field or label (displayed)
    - `string` obtain the string value of the field
    - `textColor: aColour` sets the colour of the text
    - `notification: aBlock` set a block to be executed for each action on the field. Note that `string` is normally enough to obtain the value, but this is provided for lower level control, e.g., editing. See discussion about actions and notifications.
12. `TextBox` provides an area for text entry; same as above, except created with a maximum line count.
    - `new: lines` create a new text box with a maximum number of `lines`. Or use `new` for a default of 100.
13. `Point`. Note that addition and subtraction of `Points` is implemented.
    - `anInteger @ anInteger` create using this instead of `new`.

- `x` and `y` obtain values
- `x: anInteger` and `y: anInteger` set values

14. `Font` create fonts. Note that the names are not always obvious; they can be in two parts (like `Avenir-Black` or `Avenir-Oblique`. A good list of names are in iosfonts.com. In practice this means that many times the bold and italic attributes are part of the font, and not a characteristic of it (and cannot thus be changed in the method below). However, the standard `Helvetica`, `Times Roman`, etc., work well.

    - `name: aName size: aSize bold: aBoolean italic: aBoolean` create a font with the named attributes.
    - `name: aName size: aSize` short form of the above, without bold/italic.

15. `Menu` provides access to the menu system; see discussion above

    - `title: aString` create a menu with the given title
    - `item: anItem` add a `MenuItem` to the menu
    - `menu: aMenu forItem: anItem` replaces `anItem` in a menu with another menu, providing nested menus

16. `MenuItem` create a menu item

    - `title: aString key: aKey action: aBlock` shows the title in the menu, and executes the block either on menu selection or COMMAND-key. Note that keys are case-sensitive. Thus a `key: 'A'` is activated with COMMAND-SHIFT-a, and `key: 'a'` with COMMAND-a.
    - `cutTitle: aString key: aKey` set to `cutTitle: 'Cut' key: 'X'` for standard functionality in Edit menu (but create that one too)
    - `copyTitle: aString key: aKey` set to `copyTitle: 'Copy' key: 'C'` for standard functionality in Edit menu
    - `pasteTitle: aString key: aKey` set to `pasteTitle: 'Paste' key: 'V'` for standard functionality in Edit menu
    - `selectTitle: aString key: aKey` set to `selectTitle: 'Select All' key: 'A'` for standard functionality in Edit menu

17. `Alert` provide an Alert box, with buttons

    - `title: aString message: aString style: aStyle button: aString button: aString` display the indicated message with given characteristics. The button strings are displayed on the buttons. The result of the message is an integer, 1000 for the first button (typicall an OK), 1001 for the second (typically a Cancel). The styles are as follows, and will cause the appropriate icon to be displayed

        - 0 = Warning
        - 1 = Information
        - 2 = Critical

- `title: aString message: aString style: aStyle button: aString` only one button with the indicated text.
- `title: aString message: aString style: aStyle` also one button, but the system-supplied normal MacOS OK button.
- `title: aString message: aString` the normal MacOS OK button, and style 1 (Information).

18. `Timer` provide capability to execute block asynchronously after a set time

    - `after: anInteger action: aBlock` execute block after set number of seconds.

There are a great many more methods, most private. Especially note that a method `basicXXX` should never be used unless you know exactly what you are doing.

Other improvements

Some other changes and additions have also been made to the base system.

1. `File` additional methods

    - `size` returns the size of the file
    - `readContents` reads the whole file. Assigned to an instance of ByteArray.

2. `Time` provides access to system time

    - `now` the time in microseconds
    - `asTime` decodes the above to `yyyy mm dd hh mm ss` format.
    - `timeStamp` provide a string with the time now, in `yyyy-mm-dd hh:mm:ss` format.
    - `today` provide the current date in the format `yyyy-mm-dd` .

3. `Integer` additional methods

    - `/` division, same as `quo:` .
    - `asString` string form of integer.
    - `asTime` provide string form of a `Time now` value, format `yyyy mm dd hh mm ss` .

4. `String` additional methods

    - `,` string concatenation, same as `+` .
    - `padLeftToLength: aLength with: aString` pad receiver accordingly. If `with: aString` is omitted, padding with space.
    - `padRightToLength: aLength with: aString` pad receiver accordingly. If `with: aString` is omitted, padding with space.
    - `findString: aString startingFrom: anInteger` find a substring inside another string. Returns position or 0 on failure.
    - `copyFrom: anInteger to: anInteger` copy a substring from another string based on

position.

- `split: aString` split the receiver into a `List` of strings with a separator character from `aString`. Similar to `break: aString` but handles empty strings ‘words’ correctly (i.e., two separator characters next to each other).
- `replaceString: aString withString: aString`. replace all occurrences of a string with another string in a string. Returns the new string (does not modify self).