Practical Session 4 – UI development

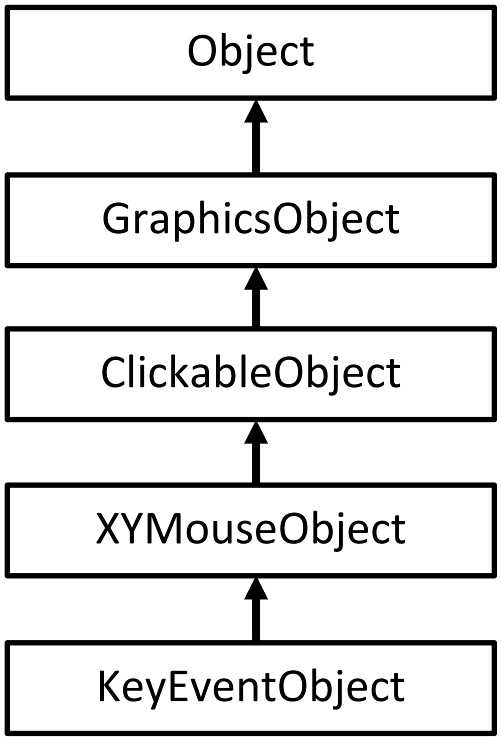
User interfaces represent some of the most complex structural systems we can build, in terms of the reference graphs needed between objects. They're a good test of a fully generalised adaptive system, and introduce you to advanced structural concepts in a component-based paradigm.

In this practical we give you time to develop your own UI-based system; it can be whatever you like.

This tutorial gives general guidance on the kinds of things you can do in Dana's standard library for GUIs.

Inheritance hierarchy

Dana's GUI framework uses an inheritance hierarchy to assist with passing notifications to different kinds of graphical object. The hierarchy looks like this:



A component can provide one of these interfaces, and when an object of that type is created and added to a window it will begin to receive notifications according to the level of the inheritance hierarchy of the interface. The GraphicsObject is a simple rendered element on the screen which is not interactive; a ClickableObject receives notifications of mouse clicks; an XYMouseObject receives notifications of mouse movements and button up/down status; and a KeyEventObject receives notifications about keyboard key presses (as well as all of the mouse notifications from the higher levels of the hierarchy). The Window interface has a function to add GraphicsObject instances; internally the window will check which sub-type an object implements and will then provide appropriate notifications to the object.

You can look up these interface types in the Dana API to see their functions in detail.

Image loading

Dana currently supports PNG images in its standard library (though it's easy to add other formats). The PNG reader is an example of *semantic variation* for interfaces. This is used when multiple components naturally provide an identical interface, but their implementations are not semantically compatible for hot-swapping. A machine learning system which is controlled component composition is then aware that it cannot replace a JPG image reader with a PNG one and still expect the system to work.

The interface used is `media.image.ImageEncoder` and the semantic variant is called `png`. You can use the API as follows:

```
uses events.EventData

component provides App requires ui.Window, media.image.ImageEncoder:png, ui.Image {

	Window wnd

	eventsink UIEvents(EventData ed) {
		if (ed.source === wnd && ed.type == Window.[close]) {
			wnd.close()
			}
		}

	int App:main(AppParam params[]) {
		wnd = new Window("PNG example")
		wnd.setSize(200, 200)
		wnd.setVisible(true)

		sinkevent UIEvents(wnd)

		ImageEncoder src = new ImageEncoder:png()
		src.loadImage("my_image.png")

		Image ix = new ui.Image(src.getPixels())
		ix.setFrameSize(180, 180)
		ix.setFrameFit(ui.Image.FIT_SCALE)
		ix.setPosition(10, 10)

		wnd.addObject(ix)

		wnd.waitForClose()

		return 0
		}
	}
```

We use the `ImageEncoder` API itself, for which we declare that we want the `png` variant, which is used to actually read data from the PNG file format into a standard representation of RGBA pixels.

Once we have this pixel map, we then use a second API called `ui.Image` which is a graphical widget for rendering pixels to the screen. This API has a set of "frame modes" which control how the image is rendered. In the above example we use the "FIT_SCALE" mode which causes the frame to scale the image down to the size of the frame and then center it in the frame.

The rest of the above code is standard windowing code to create a window and then listen for close events, exiting the program when a close event arrives.

Scroll and clipping areas

The low-level drawing API supported by the GUI framework is documented in `ui.Canvas`. Most of the operations here are self-explanatory, such as drawing lines and rectangles.

Scrolling and clipping effects are achieved with the two functions `pushSurface()` & `popSurface()` :

```
void pushSurface(Rect rect, int xscroll, int yscroll, byte alpha)
```

```
void popSurface()
```

We can "push" a surface and specify its screen coordinates via the Rect data type, which also specifies a clipping rectangle for the surface.

Any subsequent drawing commands are now *relative* to the surface (so the X coordinate 0 is the left-most edge of the surface). If anything is drawn which would go beyond the width or height of the surface, as specified in the Rect parameter, they are "clipped" so that they don't show on the screen.

When we use popSurface we then pop this surface off the stack, and any subsequent drawing is again relative to the coordinate system outside of the surface (and will no longer be clipped).

Surfaces are also the basis of scrolling elements. The pushSurface() function takes two parameters to describe the desired scroll amount in both the X (horizontal) and Y (vertical) directions, which has the effect of scrolling all of the drawing done on the surface by the given amount.

Text input

Text input is currently supported for single-line text data using the `ui.TextField` interface. This is a complex widget which deals with typing input (including keyboard mapping), mouse selection, clipboard use, context menus, and scrolling text.

It works much like any other graphical element in that you can instantiate it and then add to a window or panel, after which you can retrieve any text that has been entered into the field. A complication is that we need to map your keyboard into characters which appear in the text field. This is done via the `locale.KeyMapping` API. We currently only have one key mapping, for UK keyboards, but you could write your own implementation and then edit the `.manifest` file in the locale directory.

Mouse over/out

Graphical elements can respond to mouse clicks, but also to more detailed mouse information. If you write a graphical component which provides the XYMouseObject interface you can capture information including mouse movements and when the mouse positioned over (or "enters") your graphical object and when it is no longer over the object.

The over/out notifications can be used to support "rollover" functionality such as highlighting buttons when the mouse is over them, or changing the image shown on the screen.

The detailed X/Y mouse movement notifications can be used to support scroll bars or drag and drop.

The clipboard

You can access the operating system's clipboard (for copy and paste operations) via the `os.Clipboard` API. This allows you to insert text into the clipboard, or read text from the clipboard. Most operating systems support many different media types for clipboard contents, but we currently only support UTF8 text in Dana. If a clipboard does not have any data of this type, nothing is therefore returned if you ask for the current contents of the clipboard.