Graphical User Interfaces

Note: The exams should be corrected by next week (week beginning Monday 6th November).

User Interfaces

A user interface (in the context of software engineering) is any mechanism which allows a user to interact with a computer program or system.

An interface could be:

- Command line (CL)
- Graphical user interface (GUI)
    - Web-app
    - Native GUI

Web-app or Native GUI?

The difference between the two is narrowing now that we have HTML5, which has support for more advanced graphical features.

In the past, web-apps were more suited to CRUD type tasks, and native apps were more suited when more sophisticated interaction was required.

Now, the principles (design patterns (MVC); program architectures (event-based)) used in GUIs are being adapted in web apps too. The same principles are also at play when we develop mobile apps.

GUI Frameworks

We're taking libraries of code that others have written and constructing our GUIs from those. Those libraries handle the rendering of things on the screen, etc.

Most programming languages have support for developing GUIs:

- Java -> Swing
- Objective-C -> Cocoa
- Python -> tkinter, PyQ+, others

All of these frameworks are extensible if we want e.g. extra graphing capabilities.

All of these frameworks are based on event-based programming, where we're waiting for events to happen. Events may be button clicks, text entry, etc.

Every user interaction comes through the framework, which generates an event, we catch the event in our code and define what it means. E.g. a button is given to us, the event when the button is clicked is given to us, and we define what happens when the button is clicked.

GUI Development Tools

GUI coding creates a lot of interdependent code, and there is a lot of repetition. Adjusting layouts especially is time-consuming and technically unsophisticated.

Many tools have been created to assist with GUI development to reduce errors and reduce time cost. We will see a number of these tools, but first we need to understand the principles of what's happening under the hood.

Xcode is a development tool for OS X and iOS, and using it is very quick. 2–3 minutes vs. 40 minutes.

tkinter

tkinter is Python's interface to Tk, a library of GUI languages written to provide GUI support for TCl (which is a very old scripting language).

tkinter is a defacto standard for portable GUI development in Python (across windows, unix, and macOS.)

tkinter is extensible – numerous add-on packages such as Tix, tkk etc. provide other widgets that can be added to the standard set.

As mentioned, there are tools to ease the development burden when using tkinter.

As it is widely used, it is well-documented and there are plenty of example available.

Development Process

In tkinter, we add widgets, get them positioned onscreen where we want them to be, and register Python functions (or methods) to handle widget events. (An event is some interaction with the user, e.g. a button press.)

The functions are called "callbacks". In effect, these give meaning to the event in the context of our app. They determine what the effect of a user pressing a button will have on the state of our program.

the widgets are generic, and our callbacks make them specific to our program.

Model-View-Controller

Mode-View-Controller (MVC) is a design pattern – a standard approach to solving a common problem.

MVC separates data representations from control logic and presentation. As a result, different presentations show the same data in different ways.

It makes our code more maintainable, less error prone, and reusable.

It can be applied at different levels in the enterprise (e.g. Facebook website and Facebook app).

Model

The Model is the information being managed. This could be anything. In our code we will often use classes that we write ourselves to manage this info.

View

How the info is represented to the user (what the user sees), including data and controls.

This informs the user about the state of the model, and guides their subsequent actions (by providing controls, notifications, etc.).

Controller

What the user uses to manipulate the model.

This is the logic behind the interface controls. In tkinter we provide this using callback functions.

Example of a tkinter Program

```
from tkinter import *

root = Tk()         # Creates a toplevel container
lab = Label(root, text="Hello CS2513")      # Adds a label
lab.pack()          # Arranges or lays out widgets

root.mainloop()     # Runs the interface and handles events
```

This code creates a label, packs it, and hands control over to the framework, at which point the label is displayed to us in a window.

This window has all the standard controls, which the framework handles for us.

When we start the mainloop, we are waiting for user interactions, either mouse or key clicks. Only when an event occurs that we've defined an action for will we revert back to our code.

Analysis

- It's quite functional: we can resize it, minimise, maximise, etc.
- It's quite ugly and basic.

Sample List of Widgets

- Button – allows us to create buttons and associate them with functions
- Canvas – allows us to draw and add graphics

- Entry – allows us to enter text (text field)
- Frame – container for other widgets to assist with layout
- Menu and Menu Button – allows us to create menus

The Python documentation is a good place to look for more examples.

There are expansion packs that add more widgets, as well.

The Importance of Style

Aesthetics are very important in software.

- Users will mostly judge your program based on the user interface.
    - Our interface should look good.
- It's also the main route for erroneous data getting into our programs.
    - It should be intuitive to use, and help users understand how to get the best from the program. By doing this, we help users enter the expected values and help our code to run well.

Example with Improved Look and Feel

We'll make the interface bigger, add a title, and set a background colour.

We do this by calling `root.configure`, or using the other appropriate root method:

```
from tkinter import *

root = Tk()

root.configure(bg='lightgray')
root.minsize(height=200, width=300)

label = Label(root, text="Hello CS2513")
label.pack()
root.mainloop()
```

Layout managers invoked by pack can override these values and cause them to be ignored.

Configuring Label

We can also configure the label object (or just about any other) in the same way.

Labels can be configured by arguments to the init method, or by using the configure method.

E.g.:

```
w = Label(master, option=something, option=something, […])
```

Here `master` is the containing window, and we can specify options such as `bg` (which is for background colour).

We can also configure a label like this:

```
label['text'] = text
```

Buttons

```
button1 = Button(root, text="Turn Red", command=callback1)
```

This button will call the function `callback1` when it is clicked, and will contain the text "Turn Red".

The Entry Field

The Entry field is tkinter's text field, used for data entry. Like `input()`, it returns a string.

It's invoked similarly to other widgets:

```
entry = Entry(master, option, option, …)
```

As well as the usual configuration options, there are some that are specific to Entry, for example:

- state (Normal/Disabled) – used to enable/disable the component.
- width – the width of the text field in characters

There are also some specific methods:

- `get()` – to get the text that was set
- `delete(start, end)` – to delete the text from index position start to index position end
    - The default value for start is 0, but the default value for end is 1, so that calling `delete()` will just delete the first character.
- `insert(start, text)` – to insert text into the text field, starting at index *start*

We can also declare commands for text fields, which act if the component is interacted with. We don't usually want to do this though.

The Frame

The tkinter Frame widget is a container for other controls. We can use it to form tiles on our screen and populate these with widgets, which is useful when we have a complex layout. We can break the root frame into more discrete areas and do a particular layout for components in those areas.

Since we're restricted to using the default layout for widgets (where they stack one under the other) or specifying top, left, right, or bottom, we divide the window into frames until those layout options give us what we want.

We can assign our frames to a side and tell them to expand if they can:

```
frame1.pack(expand=True, fill="y", side="LEFT")
```

`fill="y"` restricts the frame to expanding in the y-axis.

Packing Order

When windows are shrunk, things are sacrificed according to the packing order. The first thing packed will be the last thing sacrificed.

Code Length

Our code starts to get very long as we add more components, and quickly becomes difficult to manage.

When using frames, we'll often create different classes to deal with our different frames.

Classes with GUIs

In theory, we could save a lot of code by using classes (through inheritance), but in practice there isn't enough commonality. Really, it just makes code more maintainable.

If we have an array containing a name and a location, and we want to create two labels and a button for each entry in the array, we can define a class for each row in our view.

```
from tkinter import *

class ListItem(object):

    def __init__(self, item, root):
        self._item = item
        self._frame = Frame(root)
        self._frame.pack()

        self._label1 = Label(self._frame,
text=self._item[0])
        self._label1.grid(row=1, column=1)
        self._label2 = Label(self._frame,
text=self._item[1])
        self._label2.grid(row=2, column=2)
        self._button = Button(self._frame, text="Press me",
                              command=self.handlepress)
        self._button.grid(row=1, column=3)

def handlepress(self):
    print('{0} {1}'.format(self._item[0], self._item[1]))

root = Tk()
listItems = [['Cathal'], ['Cork'],
             ['Laura'], ['Kerry'],
             ['Lucy'], ['Waterford']]

for item in listItems:
    ListItem(item, root)

root.mainloop()
```

Now we have bundled the widget with the class, which is nice. We don't need to pass any info to the `handlepress` method, as it can just access the instance info.

We can apply this to the calculator in assignment 4 if we want, but we won't lose any marks if we don't.

Master-Detail

One master view which has more basic information on a bunch of things, and if you click any particular thing, the detail view updates with more detail on that specific thing.

Through an MVC lens: The master window makes a call to the controller any time something in it is clicked, the controller works out what data to display, and puts it into the detail view.

Example with Student Class

```
from tkinter import *

class Student(object):
    pass

class Model(object):
    pass

class ListItem(object):
    def __init__(self, frame, controller, index):
        pass

class MasterView(object):
    pass

class DetailView(object):
    pass

class Controller(object):
    pass

def main():
    root = Tk()
    Controller(root)
    root.mainloop()

main()
```

- We were given a handout with this example (completed) on it.

MVC

If we use a model-view-controller design pattern, we can e.g. replace the view in order to change the layout, without altering the model or the controller.

Drawing with tkinter

tkinter provides the Canvas component to allow us to draw and animate. It's often used as the basis for graph drawing and other graphics-based tasks. It can also be used to provide animations for custom widgets that we develop.

It is created as follows:

```
can = Canvas(master, width=n, height=m)
```

Creating Objects

There are a range of methods to draw on the canvas:

```
hand = can.create_line(x0, y0, x1, y1)
```

This draws a line from (x0, y0) to (x1, y1).

```
hand = can.create_rectangle(x0, y0, x1, y1, fill='red',
outline='red')
```

This creates a rectangle with opposite corners (x0, y0) and (x1, y1).

```
hand = can.create_oval(x0, y0, x1, y1, fill='blue')
```

There are others, including creating arcs.

Deleting/Moving Objects

If we've kept a reference to an objects (`hand` in the examples above), we can delete that object like this:

```
can.delete(hand)
```

Alternatively, we can call the same `delete` method with the value `'all'` to delete everything:

```
can.delete('all')
```

We can also use the `move` canvas method to move the object to a new position:

```
canvas.move(hand, newx, newy)
```

This will delete the object from the canvas and draw it in the new position.

By using `move` with a delay time, we can achieve constant motion.

To pause the canvas for a number of milliseconds, we call `canvas.after()`:

```
canvas.after(milliseconds)
```

Example Code

```
from tkinter import *

root = Tk()
canvas = Canvas(root, width=500, height=500)
canvas.pack()

initial_x = 20
initial_y = 20

rect = canvas.create_rectangle(initial_x, initial_y,
                               initial_x + 40, initial_y +
40,
                               fill='red')

for i in range(0, 20):
    canvas.after(1000)
    canvas.move(rect, initial_x + (10 * i), initial_y + (10
* i))
    canvas.update()

root.mainloop()
```

This code causes a square to move across the screen at intervals of 1s, by a slightly larger distance each time.

Custom Events

We can also bind to user events on the canvas, for example key strokes or click events.

We can specify for instance a handler to be called on a click of the user's primary mouse button:

```
root.bind('<Button-1>', buttonpressed)
```

Now when the primary mouse button is pressed, `buttonpressed` will be called.

Updated Example

```
from tkinter import *

def buttonpressed(event):
    canvas.delete('all')
    canvas.create_rectangle(event.x, event.y,
                            event.x + 40, event.y + 40,
fill='red')

root = Tk()
canvas = Canvas(root, width=500, height=500)
canvas.pack()

initial_x = 20
initial_y = 20

rect = canvas.create_rectangle(initial_x, initial_y,
                               initial_x + 40, initial_y +
40,
                               fill='red')

root.bind('<Button-1>', buttonpressed)

root.mainloop()
```

Now every time we click, the rectangle moves to where we've clicked.

This example is on Moodle.