Advanced Programming #5: User Interfaces

CS SCHOLARS - PROGRAMMING

Learning Goals

Use **state machines** to organize all the possible interactions with an application.

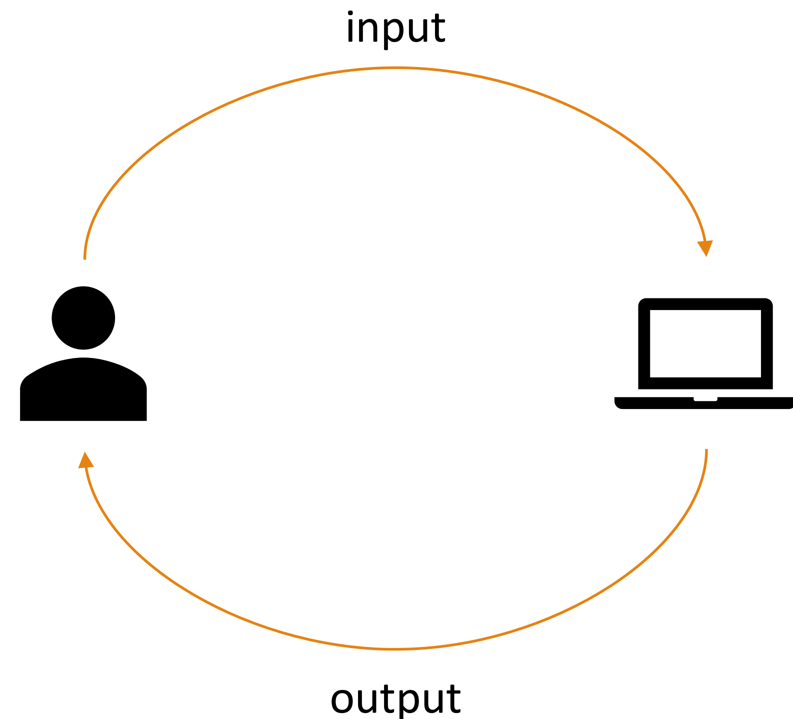
Use **widgets** to provide different kinds of inputs and outputs in applications.

User Interfaces are for Interaction

When we build an application (like an advanced game or website), we often need to build a tool that can interact with a person in **multiple different ways**.

Consider an application like an email client- the user needs to be able to see all their active messages, but also read an individual message, write a new message, and archive a single message from the list.

One way to separate out all these interactions is to design a **user interface** that lets the user focus on one task at a time, while still being able to switch tasks at need. Within a task, the interface will let the user take actions and view the results, as an advanced input/output loop.



State Machines

State Machines Organize Features

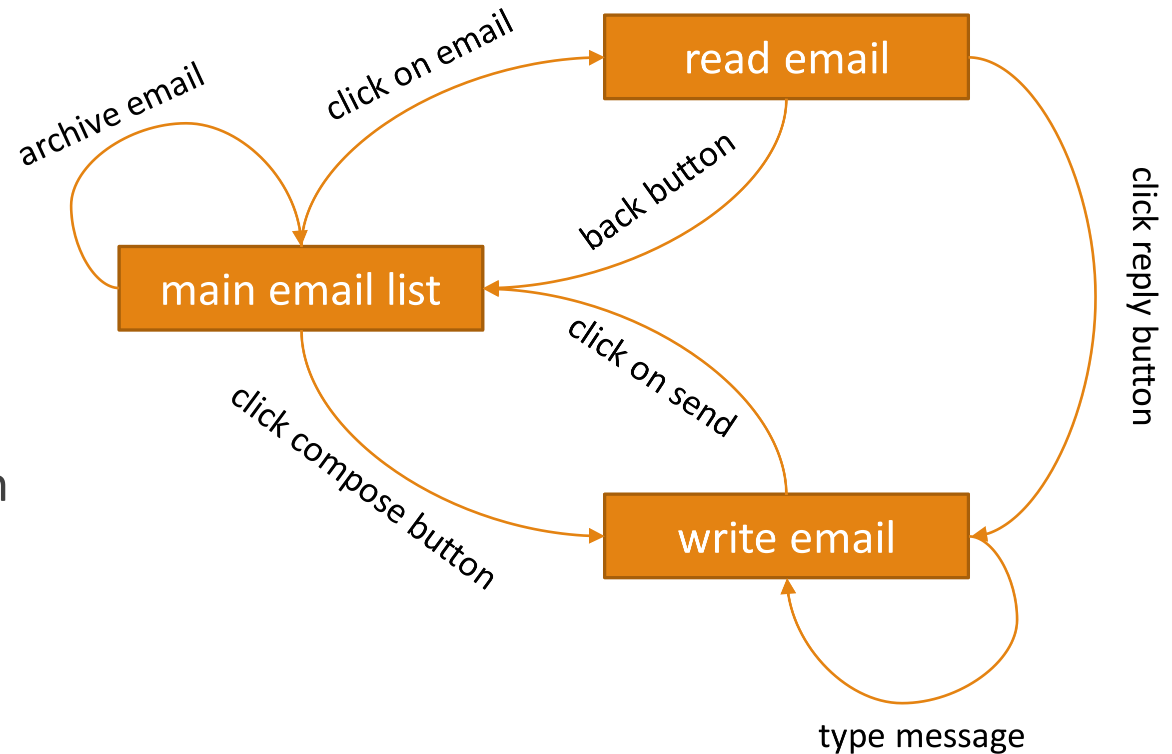
First, when designing a large or complex application, we need to break down the system into individual parts, then decide how those parts will be organized into different screens. In most cases, we won't be able to fit all tasks on one identical screen- we'll need to change the screen based on the users' needs.

We can write out **state machines** to help decide how the application will be organized *before* we start coding. This will help us determine how the code itself will be organized and connected. Planning out a design before writing the code is essential when building something large!

States and Transitions

A state machine is composed of two parts: a set of **states** (application screens where the user might end up) and **transitions** that link the states (actions that can happen which will move the user from one state to another). In other words, a state machine is just a graph!

In the email client we mentioned earlier, the state machine might look something like the graph on the right, where the rectangles are states and the arrows are transitions. Note that transitions are usually directional- you can only move from state A to state B with a certain action.



Programming State Machines

When we program a state machine, we'll represent which state we're in as part of the **model**, while transitions will all be decisions inside the **controllers**.

In the email client example, we can add a variable called **state** to our data model, then set it equal to a string that maps to the current state. We can then check that variable to see which state we're in.

Then, in `mousePressed`, we can check which state we're in, then check where the user clicked, to decide what to do next. This might look something like the code to the right.

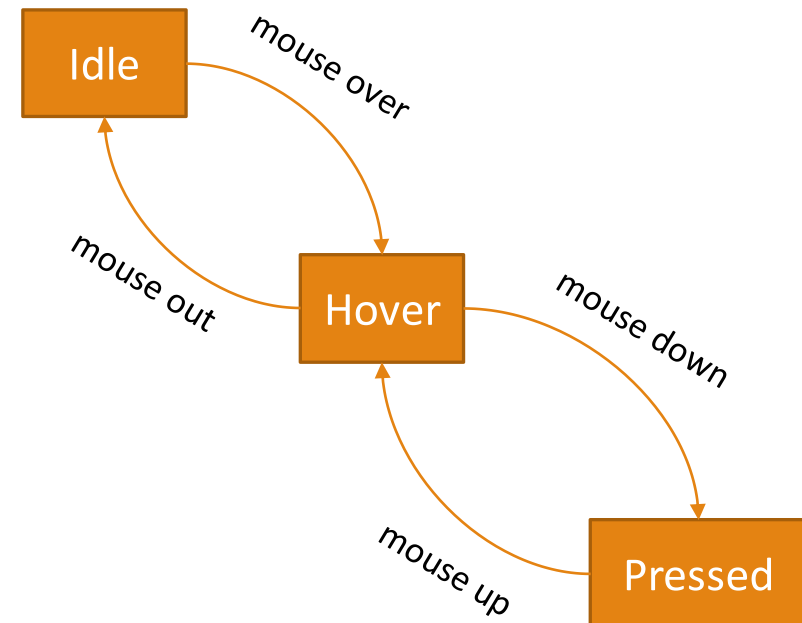
```
def init(data):
    data.state = "main"
    data.emailID = None

def mousePressed(event, data):
    if data.state == "main":
        if inArchiveButton(event.x, event.y):
            doArchive(event, data)
        elif inComposeButton(event.x, event.y):
            data.state = "write"
        elif inEmailBox(event.x, event.y):
            data.state = "read"
            data.emailID = getID(event, data)
    elif data.state == "read":
        ...
```

State Machines and Abstraction

The example we've gone into here shows how a state machine works at a high level of abstraction. But we can make state machines for low-level actions too, when needed.

Say you want to program a button that will look different based on whether the user is hovering over it with the mouse, and whether it has been clicked. The state machine for this button's appearance would look something like the example on the right.

Of course, we usually don't need to program our own button- it's provided for us!



Widgets

Widgets Provide Standard Interactions

Most graphics/layout libraries provide implementations of standard **widgets** to facilitate user interaction. A widget is a name for a component of an interface that you can find across multiple applications.

You interact with standard widgets all the time! While reading these slides you're probably using the **scrollbar** to navigate, and you might use the search **text entry** to find a specific piece of information.

Standard Widgets

While there are hundreds of widgets that we can use in interfaces, there are a subset that are used most commonly, which we'll show here.

- **Window** – the outer container for an application
- **Frame** – a box within the window that can hold and organize other widgets
- **Dialog box** – a pop-up box that asks to user to choose a button to click

- **Menu** – when hovered over, provides a list of possible actions which may themselves be menus
- **Scrollbar** – a rectangle to the side of the screen that lets the user move across a larger window

- **Label** – a box that holds pre-written text
- **Icon** – a box that holds an image

- **Button** – clicked with the mouse to start an action
- **Checkbox** – used with other checkboxes to make the user choose one or more options
- **Radio button** – used with other radio buttons to make the user choose a single option
- **Drop-down List** – when clicked, provides a box with a list of items for the user to select
- **Text box** – a box that, when clicked in, lets the user enter text with the keyboard

Widgets in Tkinter

Tkinter has most of these widgets already implemented for us to use! We can find a list of widgets with more information here:

https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html

However, to use these widgets in Tkinter, we'll need to change our simulation framework slightly. Instead of putting widgets on the canvas, we'll need to put them **directly into the root window.**

To do this, we'll need to understand our animation framework's run function...

Run – Adding Widgets

```
import tkinter

root = tkinter.Tk()

canvas = tkinter.Canvas(root, height=400, width=400)
canvas.configure(bd=0, highlightthickness=0)
canvas.pack()


canvas.create_rectangle(0, 0, 400, 500, fill="red")


button = Button(root, text="Click Me!")

button.pack()


root.mainloop()
```

The first thing we do in the run function is set `root = tkinter.Tk()`. This creates the window we'll put our application in.

To put things in the window, we need to create the widget (as we do with canvas). Note that the first argument to the Canvas call is **root**- that tells Canvas that it belongs in the root window.

Once we've finished setting up a widget, we call `widget.pack()` to actually put the widget in its parent window. The parent has a specified layout to arrange the items inside of it; by default, this is vertically from top to bottom. So when we add a Button, it will show up **under** the canvas.

Finally, when we've finished setting up everything, we call `root.mainloop()` to tell the window to stay open until we close it.

Run - Event Handlers

```
def keyEventHandler(data, canvas, event):
    keyPressed(event, data)
    canvas.delete(ALL)
    redrawAll(canvas, data)
    canvas.update()

def mouseEventHandler(data, canvas, event):
    mousePressed(event, data)
    canvas.delete(ALL)
    redrawAll(canvas, data)
    canvas.update()


    root.bind("<Key>", \
     lambda event : keyEventHandler(data, canvas, event))
    root.bind("<Button-1>", \
     lambda event : mouseEventHandler(data, canvas, event))
```

In order to make the application interactive, we need to set up **event handlers** that will capture input from the user and redirect it to our own functions. The computer is constantly monitoring input that the user creates, and it can forward that input to active applications.

In our default animation framework, we have two event handlers- `mousePressed` and `keyPressed`. In `runSimulation`, we set these up by **binding** specific events that happen within the root window to functions we define. In this case, we're binding `Button-1` (mouse click) events and `Key` events.

Note that `lambda event: ...` just lets us take the information associated with the event and send it to our own function alongside the canvas and data.

Widget Examples

Once we can pack widgets into the root window and associate them with event commands, we can start setting up real applications!

We'll go over three widget examples here: Button, Text Box, and Radio Button. We'll use each to modify the color of the canvas.

Button Example

To make a button, we need to set up the text on the button and the function that is called when we click the button.

We'll make a button that changes the color of the canvas every time we click it to a random color. We'll need to call `redrawAllWrapper` after the change to refresh the canvas.

Note that `buttonFun`'s lambda takes no arguments. That's because clicking the button doesn't generate new data. We still need to provide canvas and data so that `buttonWrapper` has access to them.

More info here:
https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/button.html

```
from tkinter import *
import random

class Struct(object): pass
data = Struct()
data.width = 400
data.height = 400
data.color = "red"
root = Tk()

def redrawAll(canvas, data):
    canvas.create_rectangle(0, 0, data.width, data.height,
                            fill=data.color)
def redrawAllWrapper(canvas, data):
    canvas.delete(ALL)
    redrawAll(canvas, data)
    canvas.update()
canvas = Canvas(root, width=data.width, height=data.height)
canvas.configure(bd=0, highlightthickness=0)
canvas.pack()

def buttonWrapper(canvas, data):
    data.color = random.choice(["red", "yellow", "green", "blue"])
    redrawAllWrapper(canvas, data)
buttonFun = lambda : buttonWrapper(canvas, data)
button = Button(root, text="Change Color", command=buttonFun)
button.pack()

redrawAllWrapper(canvas, data)
root.mainloop()
```

Text Box Example

Next, let's make a text box that lets the user enter the name of the color they want to set the canvas to.

A text box works differently from a button. We set it up as an `Entry` with just the parent window as an argument. And we can't set up an event handler in the text box- we'll need to set up a general handler in the root window instead.

Here, we'll listen for a `Return` event (when the enter key is pressed), then get the text from the textbox with `.get()`. We'll need to store the textbox in data for this to work. We can then check if the color is valid, and if it is, change the color and reset the text box.

More info here: https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/entry.html

```
...

def returnWrapper(canvas, data):
    text = data.textbox.get()
    if text in ["red", "yellow", "green", "blue"]:
        data.color = text
        data.textbox.delete(0, len(text))
    redrawAllWrapper(canvas, data)

root.bind("<Return>", lambda ignore:
          returnWrapper(canvas, data))
data.textbox = Entry(root)
data.textbox.pack()

...
```

Radio Button Example

Finally, let's set up radio buttons so that the user can select one of four color options.

Unlike buttons and text entries, we need to set up multiple Radiobuttons and connect them to each other. We do this by setting each of them to have the same **variable**, or group value. That value (`data.selection`) will update every time we click a radio button.

We then individualize the buttons with their text and value, then set up event handlers to set the color to the current value in `data.selection` whenever a button is clicked.

More info here: https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/radiobutton.html

```
...
def radioWrapper(canvas, data):
    data.color = data.selection.get()
    redrawAllWrapper(canvas, data)

data.selection = StringVar()
data.selection.set("red")
for color in ["red", "yellow", "green", "blue"]:
    b = Radiobutton(root, text=color, value=color,
                        variable=data.selection,
                        command=lambda : radioWrapper(canvas, data))
    b.pack()
...
```

Learning Goals

Use **state machines** to organize all the possible interactions with an application.

Use **widgets** to provide different kinds of inputs and outputs in applications.

Read more about state machines here: https://en.wikipedia.org/wiki/Finite-state_machine

Find more widgets here:
https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html