Human Computer Interaction

GUI Essentials

Prof. Andrew D. Bagdanov

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze
andrew.bagdanov AT unifi.it

October 6, 2017

Outline

General considerations

General considerations of Software Engineering

In software engineering, in general, there has been much said about two related principles of program organization:

The Principle of Separation of Concerns

Which states that one should divide applications into distinct features with as little overlap in functionality as possible. The important factor is minimization of interaction points to achieve high cohesion and low coupling. However, separating functionality at the wrong boundaries can result in high coupling and complexity between features even though the contained functionality within a feature does not significantly overlap.

The Principle of Single Responsibility

Which is intimately related with the Separation of Concerns, and states that each component or module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionality.

General considerations for GUIs

As we shall see, these principles are particularly important in the context of GUI design and implementation:

Separation of Concerns

In Graphical User Interfaces there is a natural division between code that displays, code that interprets user input, and code that manipulates underlying data representations. Due to the high degree of asynchronicity in GUI applications, separating these concerns makes all the difference.

Single Responsibility

Along similar lines, in GUI applications we want each segment of code (usually small fragments that setup or react to asynchronous events) to do one, narrowly defined thing – and only that one thing.

Where we will arrive

Right now I will introduce the basic structure of a GUI program and some of the standard classes and widgets with simple examples. We will see that this structure leads naturally to two more specific principles:

Inversion of Control

Which might also be called "Surrender of Control", this is related to the fact that most GUIs execute in a Main Loop outside of programmer control, instead calling user-defined code asynchronously as needed.

The Model-View-Controller (MVC) paradigm

This is a natural generalization of the principles of Separation of Concerns and Single Responsibility to the GUI world, where we typically have some model that must be both modified and visualized.

Caveats

Caveats

- In this class we use Kivy primarily as a didactic tool for illuminating the important concepts of interface design and implementation.
- As such, I will use examples of Kivy application structure and Kivy widgets.
- These are only intended as examples, and will not necessarily translate to other GUI frameworks or development environments.
- Most of the ideas I use Kivy to illustrate, however, will be present in some shape or form in other GUI frameworks.
- Especially in this first lecture, I have attempted to select the most general and transversal GUI concepts for examples.

Anatomy of a GUI

Anatomy of a GUI application

A basic skeleton

```
# Import some stuff
import kivy
from kivy.app import App
from kivy.uix.button import Label

# Define new widgets via *inheritance*.
class TestApp(App):
    def __init__(self, message='hello world'):
        App.__init__(self)       # Call base class constructor.
        self._message = message # Save message text.

    def build(self):
        return Label(text=self._message)

# Instantiate useful stuff.
app = TestApp(message='Howdy world!')

# Call the Main Loop (from which there is *no return*).
app.run()
```

The Application Class

- We are going to concentrate first on two aspects of this basic skeleton.
- First, the Application Class, which can be thought of as the main entry point for creating Kivy applications.
- Every Kivy application you create should begin by defining your own class that inherits from `kivy.app.App` and overrides specific methods to implement your desired functionality.
- The principle method to override is `kivy.app.App.build()`, which is responsible for building the application.
- In this case, building the application means:
  - Instantiating all component widgets that belongs to the application.
  - Returning the Root Widget, which is the root of the hierarchy defining your application (more on hierarchy later).
- In this simple application we have only a single widget, which is (by definition) our Root Widget.

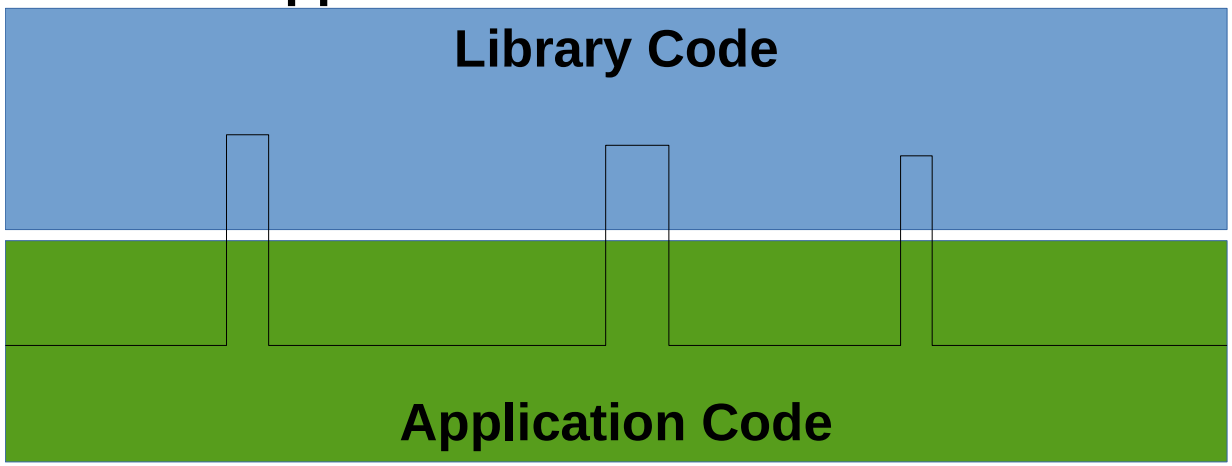
The Main Loop

- The next aspect of this program (and most GUI programs, for that matter) that you may not be familiar with is the calling of the Main Loop at the end.
- In this case, we call the `kivy.app.App.run()` method which then never returns.
- The `kivy.app.App.run()` method takes care of calling the `build()` method on our application instance object.
- It then takes care of all input handling and routing of events to and from widgets.
- This is an important difference between many software architectures and those used in GUIs: the GUI framework (which we do not control) handles the threading of program execution through the various pieces of widget code.
- Some of these pieces will be written by us, and some not.

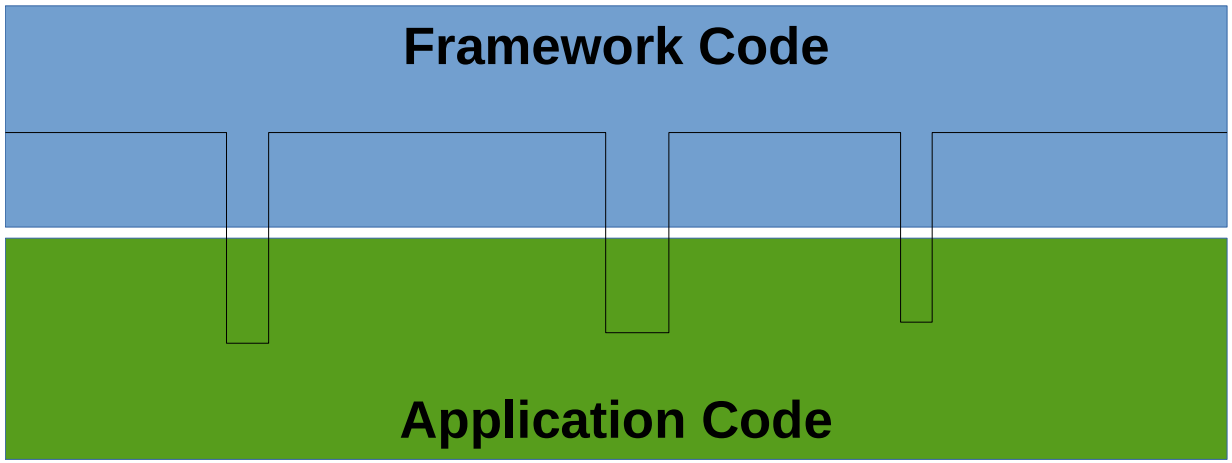
Inversion of control

This leads to the very important principle of Inversion of Control:

Standard Application Model



Inversion of Control



Inversion of control (continued)

- Inversion of Control: in a framework, unlike in libraries or normal user applications, the overall program flow of control is not dictated by the caller, but rather by the framework.
- Default behavior: a framework has a default behavior implemented by each of its components.
- Extensibility: a framework can be extended by overriding default behavior or otherwise providing user code for use by the framework.
- Non-modifiable framework code: except by overriding (e.g. by inheritance) or otherwise providing user code for use by the framework, the framework is not modifiable by the application programmer.
- In our simple example so far, we modified the default behavior of the `kivy.app.App` class by overriding the `build()` method in our derived application class.

A tour of some standard widgets

A note on python inheritance

```
class A:                                # Base class
    def __init__(self):                 # With constructor
        print 'initializing A.'

    def m1(self):
        print ' A.m1()'

class B(A):                     # Inherit from A.
    def m1(self):               # Overide m1.
        print ' B.m1()'

class C(A):                             # Inherit from A
    def __init__(self):                 # Override constructor.
        print 'initializing C.'

class D(A):                             # Make sure to call super constructor.
    def __init__(self):
        A.__init__(self)
        print 'initializing D.'
```

Application

- We already saw the `kivy.app.App` class in action in our simple example above.
- The Application class, in addition to having the Main Loop method (`run()`, in the case of Kivy), keeps track of many bits of high-level state information about the application.
- Things like the name and title of the application, the application's initialization state, and configuration options.
- The Application class (actually, our application class derived from it), is usually one of the only ways to keep track of "global" information that must be made available to other widgets.

Widget

Definition (widget)

widget, noun informal

1. a small gadget or mechanical device.
2. (in some beer cans) a plastic device which introduces nitrogen into the beer, giving it a creamy head.
3. COMPUTING: an application, or a component of an interface, that enables a user to perform a function or access a service.

- The Widget class is traditionally the base class of all widgets, or controls, available in a GUI framework.
- Often it can not be instantiated (e.g. it is pure virtual), as it is intended to be inherited from by widget designers.
- It usually holds all of the basic geometric information required by all derived widgets (e.g. position, size, etc).
- The base Widget class in Kivy is `kivy.uix.widget.Widget`

Label

- The Label widget is for rendering text (and sometimes simple graphical elements like icons or bitmaps).
- In Kivy, the `kivy.uix.label.Label` is for displaying text only and supports ascii and unicode strings.
- It also supports simple markup for rich text formatting (see laboratory exercises).

```
# hello world text
l = Label(text='Hello world')

# unicode text; can only display glyphs that are available in the font
l = Label(text=u'Hello world ' + unichr(2764))

# multiline text
l = Label(text='Multi\nLine')

# size
l = Label(text='Hello world', font_size='20sp')
```

Button

- The Button class is often derived from the Label class (and indeed it is so in Kivy).
- The `kivy.uix.button` module contains the `Button` widget.
- It has all of the textual display properties that `kivy.uix.label.Label` has (i.e. it can display rich, marked-up textual content).
- But it also inherits the `ButtonBehavior` functionality, which allows it to respond (graphically and logically) to user input (e.g. mouse and touch events).
- You can think of Button as a Label on steroids, and it might be the most used widget in any GUI framework.

Button example

```
from kivy.app import App
from kivy.uix.button import Button

class MyApp(App):
    def __init__(self, message='hello world'):
        App.__init__(self)      # Call base class constructor.
        self._message = message # Save message text.

    def build(self):
        return Button(text=self._message, padding=[50,50])

# Instantiate useful stuff.
app = MyApp(message='Howdy world!')

# Call the Main Loop (from which there is *no return*).
app.run()
```

Layouts

- Layouts are a special type of Container Widget (widgets designed to contain other widgets).
- They are how you structure the physical layout of the widgets in your application.
- Kivy supports a wide variety of layouts:
  - AnchorLayout: Widgets can be anchored to the 'top', 'bottom', 'left', 'right' or 'center'.
  - BoxLayout: Widgets are arranged sequentially, in either a 'vertical' or a 'horizontal' orientation.
  - FloatLayout: Widget positions essentially unrestricted.
  - RelativeLayout: Child widgets are positioned relative to the layout.
  - GridLayout: Widgets are arranged in a grid defined by the rows and cols properties.
  - PageLayout: Used to create simple multi-page layouts, in a way that allows easy flipping from one page to another using borders.
  - ScatterLayout: Widgets are positioned similarly to a RelativeLayout, but they can be translated, rotate and scaled.
  - StackLayout: Widgets are stacked in a lr-tb (left to right then top to bottom) or tb-lr order.

Layout example

```
# Import some stuff
from kivy.app import App
from kivy.uix.button import Button
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.scatterlayout import ScatterLayout

# Define new widgets via *inheritance*.
class TestApp(App):
    def build(self):
        layout = BoxLayout(orientation='vertical')
        buttons = [Button(text='Button {}'.format(i)) for i in range(5)]
        for b in buttons:
            layout.add_widget(b)
        return layout

# Instantiate useful stuff.
app = TestApp()
app.run()
```

Events and reacting to stuff

Events

- Events are an extremely important part of GUI programming.
- They are asynchronous occurrences that indicate that input has been received, that a property has changed, or that some other condition of interest has occurred.
- Typically, the application registers callbacks to be invoked when the event is received.
- This is precisely where the inversion of control takes place: callback code is called asynchronously by the framework whenever appropriate.
- Kivy has a rich set of events and event handling mechanisms.
- In the next lecture we will dive deeper into event handling in general, but for now we will concentrate on binding existing events to our code.

Making widgets sensitive to events

- In Kivy, we use the `bind()` method to attach executable code to events.
- For buttons, for example, we can listen for the `on_press` event:

```
# Import some stuff
from kivy.app import App
from kivy.uix.button import Button
from kivy.uix.boxlayout import BoxLayout

class TestApp(App):
    def clicked(self, instance):
        print 'Clicky clicky!'

    def build(self):
        # Create some buttons, add them to layout
        self._layout = BoxLayout(orientation='vertical')  # Vertical layout.
        buttons = [Button(text=t) for t in ['Hello world!', 'Goodbye World']]
        for b in buttons:
            b.bind(on_press=self.clicked)
            self._layout.add_widget(b)
        return self._layout

# Instantiate and run the application.
app = TestApp().run()
```

Inversion of control and separation of concerns

Inversion of control, revisited

- After this brief lecture, the concept of inversion of control should be more clear.
- Our pattern is:
  1. Derive a new application class using the default functionality of Application (`kivy.app.App`).
  2. In the `build()` method (or equivalent), construct the hierarchy of widgets that implement our interface.
  3. Create an instance of our application, and call the main loop.
- During execution of the main loop by the Kivy framework, calls will be made to the code we have provided.
- This code modifies default functionality through overriding and responding to events.

Separation of concerns, revisited

- Let's try to extrapolate based on on the simple examples we have seen today.
- Imagine not three or four widgets, but tens or hundreds, or thousands (each clickable region in Microsoft Word is an instantiated widget of some type).
- You should be able to see how the complexity of GUI application can rapidly get out of hand.
- It is for this reason that functionality must be compartmentalized and localized based on a strict separation of concerns (as much as possible).
- In fact, look at the last example and we can see a violation of this principle.

Homework

Homework

Exercise 5.1: Label formatting

Take a look at the Kivy label documentation and play around with different formatting styles available in Kivy. Use the `Label` example from this lecture as a basis, maybe with multiple `Label` widgets organized in a `BoxLayout` to demonstrate.

Exercise 5.2: Separation of concerns

Look at the last example (the `Event` example) from this lecture. What if we want to print a different message for each button? Modify the program to print a different message when the second button is clicked.

Exercise 5.3: The Kivy gallery

Spend some time browsing the Kivy Gallery to get an idea of what is possible with the Kivy UI framework.