CO664WBL CW1 Assignment

- Claudia Danciu - 22200391

In [14]:
```
# Hello Jupyter

name = input("What is your name? ")
print("Welcome to Jupyter " + name)
```

```
What is your name? Claudia
Welcome to Jupyter Claudia
```

Logbook Exercise 1

Insert a 'code' cell below. In this do the following:

- line 1 - create a list named "shopping_list" with items: milk, eggs, bread, cheese, tea, coffee, rice, pasta, milk, tea (NOTE: the duplicate items are intentional)
- line 2 - print the list along with a message e.g. "This is my shopping list ..."
- line 3 - create a tuple named "shopping_tuple" with the same items
- line 4 - print the tuple with similar message e.g. "This is my shopping tuple ..."
- line 5 - create a set named "shopping_set" from "shopping_list" by using the set() method
- line 6 - print the set with appropriate message and check duplicate items have been removed
- line 7 - make a dictionary "shopping_dict" - copy and paste the following items and prices: "milk": "£1.20", "eggs": "£0.87", "bread": "£0.64", "cheese": "£1.75", "tea": "£1.06", "coffee": "£2.15", "rice": "£1.60", "pasta": "£1.53".
- line 8 - print the dictionary with an appropriate message

An example of fully described printed output is presented below (some clues here also) Don't worry of your text output is different - it is the contents of the compund variables that matter

```
This is my shopping list ['milk', 'eggs', 'bread', 'cheese',
'tea', 'coffee', 'rice', 'pasta', 'milk', 'tea']
This is my shopping tuple ('milk', 'eggs', 'bread', 'cheese',
'tea', 'coffee', 'rice', 'pasta', 'milk', 'tea')
This is my Shopping_set with duplicates removes {'rice', 'mil
k', 'pasta', 'cheese', 'eggs', 'tea', 'bread', 'coffee'}
This is my shopping_dict {'milk': '£1.20', 'eggs': '£0.87', 'b
read': '£0.64', 'cheese': '£1.75', 'tea': '£1.06', 'coffee':
'£2.15', 'rice': '£1.60', 'pasta': '£1.53'}
```

In [2]:
```
# Creating the shopping list
shopping_list = ["milk", "eggs", "bread", "cheese", "tea", "coffee", '
print("This is my shopping list", shopping_list)

# Creating the shopping tuple
shopping_tuple = tuple(shopping_list)
print("This is my shopping tuple", shopping_tuple)

# Creating the shopping set
shopping_set = set(shopping_list)
print("This is my Shopping_set with duplicates removed", shopping_set)

# Creating the shopping dictionary
shopping_dict = {
    "milk": "£1.20",
    "eggs": "£0.87",
    "bread": "£0.64",
    "cheese": "£1.75",
    "tea": "£1.06",
    "coffee": "£2.15",
    "rice": "£1.60",
    "pasta": "£1.53"
}
print("This is my shopping_dict", shopping_dict)
```

```
This is my shopping list ['milk', 'eggs', 'bread', 'cheese', 'tea',
'coffee', 'rice', 'pasta', 'milk', 'tea']
This is my shopping tuple ('milk', 'eggs', 'bread', 'cheese', 'tea',
'coffee', 'rice', 'pasta', 'milk', 'tea')
This is my Shopping_set with duplicates removed {'cheese', 'milk',
'rice', 'tea', 'pasta', 'bread', 'coffee', 'eggs'}
This is my shopping_dict {'milk': '£1.20', 'eggs': '£0.87', 'bread':
'£0.64', 'cheese': '£1.75', 'tea': '£1.06', 'coffee': '£2.15', 'ric
e': '£1.60', 'pasta': '£1.53'}
```

Logbook Exercise 2

Create a 'code' cell below. In this do the following:

line 1 - Use a comment to title your exercise - e.g. "Unit 2 Exercise" line 2 - create a list ... li = ["USA","Mexico","Canada"] line 3 - append "Greenland" to the list l4 - print the list to de,monstrate that Greenland is attached l5 - remove "Greenland" l6 - print the list to de,monstrate that Greenland is removed l7 - insert "Greenland" at the beginning of the list l8 - print the resul of l7 l9 - shorthand slice the list to extract the first two items - simultaneausly print the output l10 - use a negative index to extract the second to last item - simultaneausly print the output l11 - use a splitting sequence to extract the middle two items - simultaneausly print the output An example of fully described printed output is presented below (some clues here also) Don't worry of your text output is different - it is the contents of the list that matter

li.append('Greenland') gives ... ['USA', 'Mexico', 'Canada', 'Greenland'] li.remove('Greenland') gives ... ['USA', 'Mexico', 'Canada'] li.insert(0,'Greenland') gives ... ['Greenland', 'USA', 'Mexico', 'Canada'] li[:2] gives ... ['Greenland', 'USA'] li[-2] gives ... Mexico li[1:3] gives ... ['USA', 'Mexico']

In [3]:
```
# Unit 2 Exercise

# Creating the list
li = ["USA", "Mexico", "Canada"]

# Appending "Greenland" to the list
li.append("Greenland")
print("li.append('Greenland') gives ... ", li)

# Removing "Greenland" from the list
li.remove("Greenland")
print("li.remove('Greenland') gives ... ", li)

# Inserting "Greenland" at the beginning of the list
li.insert(0, "Greenland")
print("li.insert(0,'Greenland') gives ... ", li)

# Shorthand slicing to extract the first two items
print("li[:2] gives ... ", li[:2])

# Using a negative index to extract the second to last item
print("li[-2] gives ... ", li[-2])

# Using slicing to extract the middle two items
print("li[1:3] gives ... ", li[1:3])
```

```
li.append('Greenland') gives ...  ['USA', 'Mexico', 'Canada', 'Green
land']
li.remove('Greenland') gives ...  ['USA', 'Mexico', 'Canada']
li.insert(0,'Greenland') gives ...  ['Greenland', 'USA', 'Mexico',
'Canada']
li[:2] gives ...  ['Greenland', 'USA']
li[-2] gives ...  Mexico
li[1:3] gives ...  ['USA', 'Mexico']
```

Logbook Exercise 3

Create a 'code' cell below. In this do the following:

on the first line create the following set ... a=[0,1,2,3,4,5,6,7,8,9,10] on the second line create the following set ... b=[0,5,10,15,20,25] on the third line create the following dictionary ... topscores={"Jo":999, "Sue":987, "Tara":960; "Mike":870} use a combination of print() and type() methods to produce the following output list a is ... [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] list b is ... [0, 5, 10, 15, 20, 25] The type of a is now ... <class 'list'> on the next 2 lines convert list a and b to sets using set() on the following lines use a combination of print(), type() and set notaion (e.g. 'a & b', 'a | b', 'b-a') to obtain the following output set a is ... {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} set b is ... {0, 5, 10, 15, 20, 25} The type of a is now ... <class 'set'> Intersect of a and b is [0, 10, 5] Union of a and b is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25] Items unique to set b are {25, 20, 15} on the next 2 lines use print(), '.keys()' and '.values()' methods to obtain the following output topscores dictionary keys are dict_keys(['Jo', 'Sue', 'Tara', 'Mike']) topscores dictionary values are dict_values([999, 987, 960, 870])

In [5]:
```
# Creating lists
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [0, 5, 10, 15, 20, 25]

# Printing list a and its type
print("list a is ... ", a)
print("list b is ... ", b)
print("The type of a is now ...", type(a))

# Converting lists to sets
a = set(a)
b = set(b)

# Printing set a and its type
print("set a is ... ", a)
print("set b is ... ", b)
print("The type of a is now ...", type(a))

# Performing set operations
print("Intersect of a and b is", a & b)
print("Union of a and b is", a | b)
print("Items unique to set b are", b - a)
# Creating the dictionary
topscores = {"Jo": 999, "Sue": 987, "Tara": 960, "Mike": 870}

# Printing dictionary keys and values
print("topscores dictionary keys are", topscores.keys())
print("topscores dictionary values are", topscores.values())
```

```
list a is ...  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list b is ...  [0, 5, 10, 15, 20, 25]
The type of a is now ... <class 'list'>
set a is ...  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
set b is ...  {0, 5, 10, 15, 20, 25}
The type of a is now ... <class 'set'>
Intersect of a and b is {0, 10, 5}
Union of a and b is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25}
Items unique to set b are {25, 20, 15}
topscores dictionary keys are dict_keys(['Jo', 'Sue', 'Tara', 'Mik
e'])
topscores dictionary values are dict_values([999, 987, 960, 870])
```

Logbook Exercise 4

Create a 'code' cell below. In this do the following:

Given the following 4 lists of names, house number and street addresses, towns and postcodes ... ["T Cruise","D Francis","C White"] ["2 West St","65 Deadend Cls","15 Magdalen Rd"] ["Canterbury", "Reading", "Oxford"] ["CT8 23RD", "RG4 1FG", "OX4 3AS"] write a Custom 'address_machine' function that formats 'name', 'hs_number_street', 'town', 'postcode' with commas and spaces between items create a 'newlist' that repeatedly calls 'address_machine' and 'zips' items from the 4 lists write a 'for loop' that iterates over 'new list' and prints each name and address on a separate line the output should appear as follows T Cruise, 2 West St, Canterbury, CT8 23RD D Francis, 65 Deadend Cls, Reading, RG4 1FG C White, 15 Magdalen Rd, Oxford, OX4 3AS HINT: look at "# CUSTOM FUNCTION WORKED EXAMPLES 3 & 4" above

In [18]:
```
# Custom function to format address
def address_machine(name, hs_number_street, town, postcode):
    return f"{name}, {hs_number_street}, {town}, {postcode}"

# Given lists of data
names = ["T Cruise", "D Francis", "C White"]
house_numbers_streets = ["2 West St", "65 Deadend Cls", "15 Magdalen F
towns = ["Canterbury", "Reading", "Oxford"]
postcodes = ["CT8 23RD", "RG4 1FG", "OX4 3AS"]

# Creating new list by zipping items and applying address_machine func
newlist = [address_machine(name, hs_number_street, town, postcode) for

# Iterating over new list and printing each name and address on a sepa
for item in newlist:
    print(item)

    # Adding an empty row for readability
    print()
```

```
T Cruise, 2 West St, Canterbury, CT8 23RD

D Francis, 65 Deadend Cls, Reading, RG4 1FG

C White, 15 Magdalen Rd, Oxford, OX4 3AS
```

Logbook Exercise 5

Create a 'code' cell below. In this do the following:

Create a super class "Person" that takes three string and one integer parameters for first and second name, UK Postcode and age in years. Give "Person" a method "greeting" that prints a statement along the lines "Hello, my name is Freddy Jones. I am 22 years old and my postcode is HP6 7AJ" Create a "Student" class that extends/inherits "Person" and takes additional parameters for degree_subject and student_ID. give "Student" a "studentGreeting" method that prints a statement along the lines "My student ID is SN123456 and I am reading Computer Science" Use either Python {} format or C-type %s/%d notation to format output strings Create 3 student objects and persist these in a list Iterate over the three objects and call their "greeting" and "studentGreeting" methods Output should be along the lines of the following Hello, my name is Dick Turpin. I am 32 years old and my postcode is HP11 2JZ My student ID is DT123456 and I am reading Highway Robbery

Hello, my name is Dorothy Turpin. I am 32 years old and my postcode is SO14 7AA My student ID is DT123457 and I am reading Law

Hello, my name is Oliver Cromwell. I am 32 years old and my postcode is OX35 14RE My student ID is OC123456 and I am reading History

In [19]:
```
# Super class definition
class Person:
    def __init__(self, first_name, last_name, postcode, age):
        self.first_name = first_name
        self.last_name = last_name
        self.postcode = postcode
        self.age = age

    def greeting(self):
        print("Hello, my name is {0} {1}. I am {2} years old and my pc

# Subclass definition
class Student(Person):
    def __init__(self, first_name, last_name, postcode, age, degree_su
        super().__init__(first_name, last_name, postcode, age)
        self.degree_subject = degree_subject
        self.student_ID = student_ID

    def studentGreeting(self):
        print("My student ID is {0} and I am reading {1}".format(self.

# Creating student objects
student1 = Student("Dick", "Turpin", "HP11 2JZ", 32, "Highway Robbery'
student2 = Student("Dorothy", "Turpin", "SO14 7AA", 32, "Law", "DT1234
student3 = Student("Oliver", "Cromwell", "OX35 14RE", 32, "History", '

# Persisting student objects in a list
students = [student1, student2, student3]

# Iterating over student objects and calling their methods
for student in students:
    student.greeting()
    student.studentGreeting()

    # Adding an empty row for readability
    print()
```

```
Hello, my name is Dick Turpin. I am 32 years old and my postcode is
HP11 2JZ
My student ID is DT123456 and I am reading Highway Robbery

Hello, my name is Dorothy Turpin. I am 32 years old and my postcode
is SO14 7AA
My student ID is DT123457 and I am reading Law

Hello, my name is Oliver Cromwell. I am 32 years old and my postcode
is OX35 14RE
My student ID is OC123456 and I am reading History
```

Logbook Exercise 6

Examine Steve Lipton's "simplest ever" version for the MVC Note how when the MyController object is initialised it: passes a reference to itself to the MyModel and MyView objects it creates thereby allowing MyModel and MyView to create 'delegate' vc (virtual control) aliases to call back the MyController object When you feel you have understood MVC messaging and delegation add code for a new button that removes items from the list

The end result should be capable of creating the output below Clearly comment your code to highlight the insertions you have made Note: if you don't see the GUI immediately look for the icon Jupyter icon in your task bar (also highlighted below) mvc_task.png

In [20]:
```
# Import necessary libraries
from tkinter import *

# Controller class
class MyController:
    def __init__(self, parent):
        self.parent = parent
        self.model = MyModel(self)
        self.view = MyView(self)
        self.view.setEntry_text('Add to Label')
        self.view.setLabel_text('Ready')

    # Event handler for quit button
    def quitButtonPressed(self):
        self.parent.destroy()

    # Event handler for add button
    def addButtonPressed(self):
        self.view.setLabel_text(self.view.entry_text.get())
        self.model.addToList(self.view.entry_text.get())

    # Event handler for remove button
    def removeButtonPressed(self):
        self.model.removeFromList()

    # Delegate method for list change
    def listChangedDelegate(self):
        print(self.model.getList())

# View class
class MyView(Frame):
    def __init__(self, vc):
        self.frame = Frame()
        self.frame.grid(row=0, column=0)
        self.vc = vc
        self.entry_text = StringVar()
        self.entry_text.set('nil')
        self.label_text = StringVar()
        self.label_text.set('nil')
        self.loadView()

    def loadView(self):
        quitButton = Button(self.frame, text='Quit', command=self.vc.q
        addButton = Button(self.frame, text="Add", command=self.vc.add
        removeButton = Button(self.frame, text="Remove", command=self.
        entry = Entry(self.frame, textvariable=self.entry_text).grid(r
        label = Label(self.frame, textvariable=self.label_text).grid(r

    def getEntry_text(self):
        return self.entry_text.get()

    def setEntry_text(self, text):
        self.entry_text.set(text)

    def getLabel_text(self):
        return self.label_text.get()

    def setLabel_text(self, text):
        self.label_text.set(text)

# Model class
```

```
class MyModel:
    def __init__(self, vc):
        self.vc = vc
        self.myList = ['duck', 'duck', 'goose', 'penguin', 'chicken',
        self.count = 0

    # Delegate method for list change
    def listChanged(self):
        self.vc.listChangedDelegate()

    # Method to get the list
    def getList(self):
        return self.myList

    # Method to add an item to the list
    def addToList(self, item):
        self.myList.append(item)
        self.listChanged()

    # Method to remove an item from the list
    def removeFromList(self):
        if self.myList:
            self.myList.pop()
            self.listChanged()

# Main function to run the application
def main():
    root = Tk()
    frame = Frame(root, bg='#0555ff')
    root.title('Hello Penguins')
    app = MyController(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

```
['duck', 'duck', 'goose', 'penguin', 'chicken', 'turkey', 'pheasan
t', 'rabbit']
```

Logbook Exercise 7

Your task is to extend the Observer example below with a pie-chart view of model data and to copy this cell and the solution to your logbook The bar chart provides a useful example of structure Partial code is provided below for insertion, completion (note '####' requires appropriate replacement) and implementation you will also need to create an 'observer' object from the PieView class and attach it to the first 'model'

Pie chart viewer/ConcreteObserver - overrides the update() method

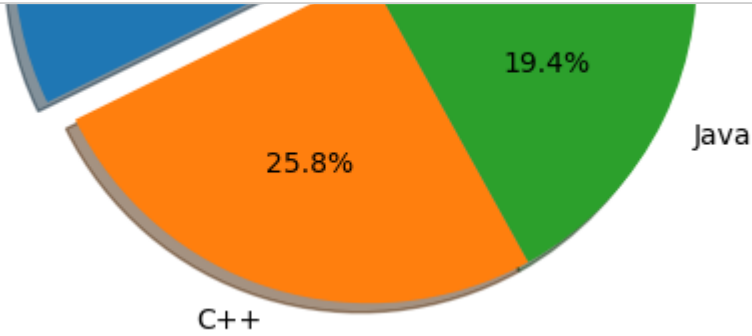
class PieView(####):

```
def update(####, ####): #Alert method that is invoked when the
notify() method in a concrete subject is invoked
    # Pie chart, where the slices will be ordered and plotted
counter-clockwise:
    labels = ####
    sizes = ####
    explode = (0.1, 0, 0, 0, 0, 0)  # only "explode" the 1st s
lice
    fig1, ax1 = plt.subplots()
    ax1.pie(sizes, explode=explode, labels=labels, autopct='%
1.1f%%', shadow=True, startangle=90)
```

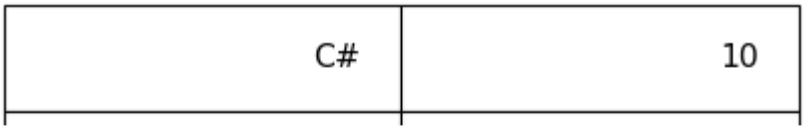
In [6]:
```
import matplotlib.pyplot as plt
import numpy as np


# Nicely abstracted structure by which any model can notify any observ
class Subject(object): #Represents what is being 'observed'

    def __init__(self):
        self._observers = [] # This where references to all the observ
                             # Note that this is a one-to-many relatio

    def attach(self, observer):
        if observer not in self._observers: #If the observer is not al
            self._observers.append(observer) # append the observer to

    def detach(self, observer): #Simply remove the observer
        try:
            self._observers.remove(observer)
        except ValueError:
            pass

    def notify(self, modifier=None):
        for observer in self._observers: # For all the observers in th
            if modifier != observer: # Don't notify the observer who i
                observer.update(self) # Alert the observers!

# Represents the 'data' for which changes will produce notifications t
class Model(Subject): # Extends the Subject class

    def __init__(self, name):
        Subject.__init__(self)
        self._name = name # Set the name of the model
        self._labels = ['Python', 'C++', 'Java', 'Perl', 'Scala', 'Lis
        self._data = [10,8,6,4,2,1]

    @property #Getter that gets the labels
    def labels(self):
        return self._labels

    @labels.setter #Setter that sets the labels
    def labels(self, labels):
        self._labels = labels
        self.notify() # Notify the observers whenever somebody changes

    @property #Getter that gets the data
    def data(self):
        return self._data

    @data.setter #Setter that sets the labels
    def data(self, data):
        self._data = data
        self.notify() # Notify the observers whenever somebody changes


# This is the 'standard' view/observer which also acts as an 'abstract
class View():

    def __init__(self, name=""):
         self._name = name #Set the name of the Viewer

    def update(self, subject): #Alert method that is invoked when the
```

```
        print("Generalised Viewer '{}' has: \nName = {}; \nLabels = {}

# Table 'chart' viewer/ConcreteObserver - overrides the update() metho
class TableView(View):

    def update(self, subject): #Alert method that is invoked when the
        fig = plt.figure(dpi=80)
        ax = fig.add_subplot(1,1,1)
        table_data = list(map(list,zip(subject._labels, subject._data)
        table = ax.table(cellText=table_data, loc='center')
        table.set_fontsize(14)
        table.scale(1,4)
        ax.axis('off')
        plt.show()

# Bar chart viewer/ConcreteObserver - overrides the update() method
class BarView(View):

    def update(self, subject): #Alert method that is invoked when the
        objects = subject._labels
        y_pos = np.arange(len(objects))
        performance = subject._data
        plt.bar(y_pos, performance, align='center', alpha=0.5)
        plt.xticks(y_pos, objects)
        plt.ylabel('Usage')
        plt.title('Programming language usage')
        plt.show()

# Pie chart viewer/ConcreteObserver - overrides the update() method
class PieView(View):

    def update(self, subject): #Alert method that is invoked when the
        labels = subject._labels
        sizes = subject._data
        explode = (0.1, 0, 0, 0, 0, 0)  # only "explode" the 1st slice
        fig1, ax1 = plt.subplots()
        ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%
        ax1.axis('equal')  # Equal aspect ratio ensures that pie is dr
        plt.title('Programming language usage')
        plt.show()

#Creating the subjects
m1 = Model("Model 1")
m2 = Model("Model 2") # This is never used!

#Creating the observers
v1 = View("1: standard text viewer")
v2 = TableView("2: table viewer")
v3 = BarView("3: bar chart viewer")
v4 = PieView("4: pie chart viewer")
####

#Attaching the observers to the first model
m1.attach(v1)
m1.attach(v2)
m1.attach(v3)
m1.attach(v4)
####

# Calling the notify() method to see all the charts in their unchanged
m1.notify()
```

```
# Changing the properties of the first model
# Changing 1 triggers all 4 views and updates their labels
m1.labels = ['C#','PHP','JavaScript','ASP','Python','Smalltalk']
# Change 2 triggers all 4 views and updates their data
m1.data = [1,18,8,60,3,1]
```




```
Generalised Viewer '1: standard text viewer' has:
Name = Model 1;
Labels = ['C#', 'PHP', 'JavaScript', 'ASP', 'Python', 'Smalltalk'];
Data = [10, 8, 6, 4, 2, 1]
```



Logbook Exercise 8

Your task is to extend the modified version of Burkhard Meier's button factory (below) to create a text field factory In tkinter textfields are 'Entry' widgets Similary to the button factory structure you will need: a concrete tk.Entry widget factory class - name it TextFactory() the TextFactory's Factory Method - name it create_text(...) an abstract product - name it TextBase() and give it default attributes 'textvariable' and 'background' ... the constructor code for the tk.Entry widgets will be something like tk_string = tk.StringVar() tk_string.set(self.textvariable) super().**init**(frame, textvariable=tk_string, background=self.background, foreground="white") self.grid(column=1, row=row) 3x concrete text products

... and assign them textvariable values of 'red type', 'blue type', "green type' respectively

... and assign them background values of 'red', 'blue', and 'green' respectively

To extend the OOP class with a create_text_fields() method

... that creates a factory object

... and the factory method will need to be called to create each text widget

the end product should look something like this ...

In [7]:
```
'''
Example modified from Meier, B (2017) Python GUI Programming Cookbook,

Refactored by Justin Cross November 2023

'''

import tkinter as tk
from tkinter import ttk

class ButtonBase(tk.Button):
    title      = 'Button'
    relief     = 'flat'
    foreground = 'white'

    # The constructor requires the LabelFrame object where the button
    def __init__(self, frame, row):
        # Calling the tk.Button constructor passing in the required pa
        super().__init__(frame, text=self.title, relief=self.relief, f
        self.grid(column=0, row=row, sticky='ew', padx=5, pady=5)

# Concrete Product buttons
class ButtonRidge(ButtonBase):
    title      = 'Button 1'
    relief     = 'ridge'
    foreground = 'red'

class ButtonSunken(ButtonBase):
    title      = 'Button 2'
    relief     = 'sunken'
    foreground = 'blue'

class ButtonGroove(ButtonBase):
    title      = 'Button 3'
    relief     = 'groove'
    foreground = 'green'

class ButtonFactory():
    _button_classes = [ButtonRidge, ButtonSunken, ButtonGroove]

    # The factory method
    def create_button(self, frame, index):
        # Getting the class for the type of button needed to be create
        button_class = self._button_classes[index]
        # Calling the constructor of the class with the required frame
        return button_class(frame, index)

class TextBase(tk.Entry):
    textvariable = ''
    background = ''

    def __init__(self, frame, row, textvariable, background):
        tk_string = tk.StringVar(value=textvariable)
        super().__init__(frame, textvariable=tk_string, foreground="wh
        # Set the default text for the text entry
        tk_string.set(self.textvariable)
        self.grid(column=1, row=row, sticky='ew', padx=5, pady=5)

# Concrete text entry classes with specific properties

class RedText(TextBase):
```

```
    textvariable = 'red type'
    background = 'red'

class BlueText(TextBase):
    textvariable = 'blue type'
    background = 'blue'

class GreenText(TextBase):
    textvariable = 'green type'
    background = 'green'

# Text factory for creating text entry instances

class TextFactory():
    _text_classes = [RedText, BlueText, GreenText]

    def create_text(self, frame, index):
        text_class = self._text_classes[index]
        return text_class(frame, index, text_class.textvariable, text_

class OOP():
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Python GUI")
        self.create_widgets()

    def create_widgets(self):
        self.frame = ttk.LabelFrame(text=' Widget Factory ')
        self.frame.grid(padx=40, pady=10)

        self.create_buttons_and_texts()

    def create_buttons_and_texts(self):
        button_factory = ButtonFactory()
        text_factory = TextFactory()
        for i in range(3):
            button = button_factory.create_button(self.frame, i)
            text = text_factory.create_text(self.frame, i)
            # Making sure the grid call is separate from the construct
            button.grid(column=0, row=i, padx=5, pady=5, sticky='ew')
            text.grid(column=1, row=i, padx=5, pady=5, sticky='ew')

        # Setting the column configuration to allow text entries to ex
        self.frame.columnconfigure(1, weight=1)

#==========================
oop = OOP()
oop.window.mainloop()
```

Logbook Exercise 9

- Extend the Jungwoo Ryoo's Abstract Factory below to mirror the structure used by a statically typed languages by:
- adding a 'CatFactory' and a 'Cat' class with methods that are compatible with 'DogFactory' and 'Dog' respectively
- providing an Abstract Factory class/interface named 'AnimalFactory' and make both the Dog and Cat factories implement this

- providing an AbstractProduct (name this 'Animal') and make both Dog and cat classes implement this
- Use in-code comments (#) to identify the abstract and concrete entities present in Gamma et al. (1995)
- comments should include: "# Abstract Factory #"; "# Concrete Factory #"; "# Abstract Product #"; "# Concrete Product #"; and "# The Client #"
- Implement the CatFactory ... the end output should look something like this ...

```
Our pet is 'Dog'!
Our pet says hello by 'Woof'!
Its food is 'Dog Food'!

Our pet is 'Cat'!
Our pet says hello by 'Meeoowww'!
```

In [9]:
```
# Abstract Factory #
class AnimalFactory:
    def get_pet(self):
        pass
    def get_food(self):
        pass

# Concrete Factory #
class DogFactory(AnimalFactory):
    def get_pet(self):
        """Returns a Dog object"""
        return Dog()
    def get_food(self):
        """Returns a Dog Food object"""
        return "Dog Food!"

# Concrete Factory #
class CatFactory(AnimalFactory):
    def get_pet(self):
        """Returns a Cat object"""
        return Cat()
    def get_food(self):
        """Returns a Cat Food object"""
        return "Cat Food!"

# Abstract Product #
class Animal:
    def speak(self):
        pass
    def __str__(self):
        pass

# Concrete Product #
class Dog(Animal):
    """One of the objects to be returned"""
    def speak(self):
        return "Woof!"
    def __str__(self):
        return "Dog"

# Concrete Product #
class Cat(Animal):
    """One of the objects to be returned"""
    def speak(self):
        return "Meeoowww"
    def __str__(self):
        return "Cat"

# The Client #
class PetStore:
    """PetStore houses our Abstract Factory"""
    def __init__(self, pet_factory=None):
        """pet_factory is our Abstract Factory"""
        self._pet_factory = pet_factory

    def show_pet(self):
        """Utility method displays details of objects returned by DogF
        pet = self._pet_factory.get_pet()
        pet_food = self._pet_factory.get_food()
        print("Our pet is '{}'!".format(pet))
        print("Our pet says hello by '{}'".format(pet.speak()))
```

```
        print("Its food is '{}'!".format(pet_food))

# Creating a Concrete Factory for Dog
dog_factory = DogFactory()

# Creating a pet store housing the Abstract Factory for Dog
shop = PetStore(dog_factory)

# Invoking the utility method to show the details of the pet Dog
shop.show_pet()

# Adding extra empty line for readability
print()

# Creating a Concrete Factory for Cat
cat_factory = CatFactory()

# Creating a pet store housing the Abstract Factory for Cat
shop = PetStore(cat_factory)

# Invoking the utility method to show the details of the pet Cat
shop.show_pet()
```

```
Our pet is 'Dog'!
Our pet says hello by 'Woof!'
Its food is 'Dog Food!'!

Our pet is 'Cat'!
Our pet says hello by 'Meeoowww'
Its food is 'Cat Food!'!
```

Logbook Exercise 10

- Modify Jungwoo Ryoo's Strategy Pattern to showcase ***OpenCV*** capabilities with different image processing strategies
- We will use the ***OpenCV*** (Open Computer Vision) library which has been reproduced with Python bindings
- OpenCV has many standard computer science image-processing filters and includes powerful AI machine learning algorithms
- The following resources provide more information on OpenCv with Python ...
- Beyeler, M. (2015). OpenCV with Python blueprints. Packt Publishing Ltd.
- Joshi, P. (2015). OpenCV with Python by example. Packt Publishing Ltd.
- The cartoon effect is from http://www.askaswiss.com/2016/01/how-to-create-cartoon-effect-opencv-python.html (http://www.askaswiss.com/2016/01/how-to-create-cartoon-effect-opencv-python.html) and https://www.tutorialspoint.com/cartooning-an-image-using-opencv-in-python (https://www.tutorialspoint.com/cartooning-an-image-using-opencv-in-python)

Development stages

- Install the OpenCV package - we have to do this manually ...
- Start Anaconda Navigator
- From Anaconda run the CMD.exe terminal
- At the prompt type ... `conda install opencv-python`

- The process may pause with a prompt ... `Proceed ([y]/n)?` ... just accept this ... `y`
- Copy Jungwoo Ryoo's code to a code cell below this one
- As well as the ***types*** module you will need to provide access to OpenCV and numpy as follows

```
# Import OpenCV
import cv2
import numpy as np
```

- Please place a copy of clouds.jpg in the same directory as your Jupyter logbook
- The code for each strategy and some notes on implementing these are below ...
- The output should look something like this ... but if you wish feel free to experiment with something else ... cats etc.!

Implementing image processing strategies

- There will be six strategy objects s0-s5, where s0 is the default strategy of the **Strategy** class
- Instead of assigning a name to each strategy object, you will need to reference the image to be processed - 'clouds.jpg'
- i.e. s0.image = "clouds.jpg"
- The *body* code for each strategy is below, you will need to provide the method signatures and their executions

strategy s0

The default ***execute()*** method that simply displays the image sent to it

```
        print("The image {} is used to execute Strategy 0 – Di
splay image".format(self.image))
        img_rgb = cv2.imread(self.image)
        cv2.imshow('Image', img_rgb)
```

strategy s1

This converts a colour image into a mononochrome one - suggested strategy method name is ***strategy_greyscale***

```
img_rgb = cv2.imread(self.image)
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
cv2.imshow('Greyscale image', img_gray)
```

strategy s2

This applies a blur filter to an image - suggested strategy method name is ***strategy_blur***

```
img_rgb = cv2.imread(self.image)
img_blur = cv2.medianBlur(img_rgb, 7)
cv2.imshow('Blurred image', img_blur)
```

strategy s3

This produces a colour negative image from a colour one - suggested strategy method name is ***strategy_colNegative***

```
  img_rgb = cv2.imread(self.image)
    for x in np.nditer(img_rgb, op_flags=['readwrite']):
        x[...] = (255 - x)
    cv2.imshow('Colour negative', img_rgb)
```

strategy s4

This produces a monochrome negative image from a colour one - suggested strategy method name is ***strategy_greyNegative***

```
    img_rgb = cv2.imread(self.image)
    img_grey = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    for x in np.nditer(img_grey, op_flags=['readwrite']):
        x[...] = (255 - x)
    cv2.imshow('Monochrome negative', img_grey)
```

strategy s5

This produces a cartoon-like effect - suggested strategy method name is ***strategy_cartoon***

```
    #Use bilateral filter for edge smoothing.
    num_down = 2 # number of downsampling steps
    num_bilateral = 7 # number of bilateral filtering steps
    img_rgb = cv2.imread(self.image)
    # downsample image using Gaussian pyramid
    img_color = img_rgb
    for _ in range(num_down):
        img_color = cv2.pyrDown(img_color)
    # repeatedly apply small bilateral filter instead of apply
ing one large filter
    for _ in range(num_bilateral):
        img_color = cv2.bilateralFilter(img_color, d=9, sigmaC
olor=9, sigmaSpace=7)
    # upsample image to original size
    for _ in range(num_down):
        img_color = cv2.pyrUp(img_color)
    #Use median filter to reduce noise convert to grayscale an
d apply median blur
    img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    img_blur = cv2.medianBlur(img_gray, 7)
    #Use adaptive thresholding to create an edge mask detect a
nd enhance edges
    img_edge = cv2.adaptiveThreshold(img_blur, 255, cv2.ADAPTI
VE_THRESH_MEAN_C, cv2.THRESH_BINARY, blockSize=9, C=2)
    # Combine color image with edge mask & display picture, co
nvert back to color, bit-AND with color image
    img_edge = cv2.cvtColor(img_edge, cv2.COLOR_GRAY2RGB)
    img_cartoon = cv2.bitwise_and(img_color, img_edge)
    # display
    cv2.imshow("Cartoon-ised image", img_cartoon); cv2.waitKey
```

In [ ]:
```
import cv2
import numpy as np

# Loading the original image
image_path = 'clouds.jpg'  # Make sure to use the correct path to your
original_image = cv2.imread(image_path)

# Checking if the image was loaded correctly
if original_image is None:
    print("Error: The image cannot be loaded. Please check the file pa
    exit()

# Grayscale effect
def convert_to_grayscale(image):
    return cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Blurred effect
def apply_blur(image):
    return cv2.GaussianBlur(image, (21, 21), sigmaX=0, sigmaY=0)

# Color negative effect
def color_negative(image):
    return 255 - image

# Monochrome negative effect
def monochrome_negative(image):
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    return 255 - gray_image

# Cartoon effect
def cartoonize_image(image):
    # Convert to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # Apply median blur
    gray_blur = cv2.medianBlur(gray, 5)
    # Use adaptive thresholding to create an edge mask
    edges = cv2.adaptiveThreshold(gray_blur, 255, cv2.ADAPTIVE_THRESH_
    # Apply bilateral filter to get a cartoon effect
    color = cv2.bilateralFilter(image, 9, 300, 300)
    # Combine edges and color
    cartoon = cv2.bitwise_and(color, color, mask=edges)
    return cartoon

# Apply the effects
grayscale_image = convert_to_grayscale(original_image)
blurred_image = apply_blur(original_image)
color_negative_image = color_negative(original_image)
monochrome_negative_image = monochrome_negative(original_image)
cartoon_image = cartoonize_image(original_image)

# Display the images
cv2.imshow('Original image', original_image)
cv2.imshow('Grayscale image', grayscale_image)
cv2.imshow('Blurred image', blurred_image)
cv2.imshow('Color negative', color_negative_image)
cv2.imshow('Monochrome negative', monochrome_negative_image)
cv2.imshow('Cartoon-ised image', cartoon_image)

# Wait for a key press and then close all windows
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Logbook Exercise 11

- Demonstrate the use of `__iter__()`, `__next()__` and `StopIteration` using ...
- ... the first four items from the top10books list (see above) ...
- ... and the following structure

```
mylist = ['item1', 'item2', 'item3']

iter_mylist = iter(mylist)

try:
    print( next(iter_mylist))
    print( next(iter_mylist))
    print( next(iter_mylist))
    # Exceeds numbe of items so should raise StopIteration exc
eption
    print( next(iter_mylist))
except Exception as e:
    print(e)
    print(sys.exc_info())
```

In [1]:
```
class TopBooksIterator:
    def __init__(self, books):
        self.books = books
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < 4:  # Limit to the first four items
            result = self.books[self.index]
            self.index += 1
            return result
        else:
            raise StopIteration

# Defining a list of top books as mentioned in the instructions above
top10books = [
    "Anna Karenina by Leo Tolstoy",
    "Madame Bovary by Gustave Flaubert",
    "War and Peace by Leo Tolstoy",
    "Lolita by Vladimir Nabokov",
    "The Adventures of Huckleberry Finn by Mark Twain",
    "Hamlet by William Shakespeare",
    "The Great Gatsby by F. Scott Fitzgerald",
    "In Search of Lost Time by Marcel Proust",
    "The Stories of Anton Chekhov by Anton Chekhov",
    "Middlemarch by George Eliot"
]

# Creating the iterator for the top10books list
iter_top_books = TopBooksIterator(top10books)

# Iterating over the list using the try-except block
try:
    print(next(iter_top_books))  # Anna Karenina by Leo Tolstoy
    print(next(iter_top_books))  # Madame Bovary by Gustave Flaubert
    print(next(iter_top_books))  # War and Peace by Leo Tolstoy
    print(next(iter_top_books))  # Lolita by Vladimir Nabokov

    # The next call should exceed the number of items and raise a Stop
    print(next(iter_top_books))
except StopIteration as e:
    print("Reached the end of the iterator.")
except Exception as e:
    import sys
    print(e)
    print(sys.exc_info())
```

```
Anna Karenina by Leo Tolstoy
Madame Bovary by Gustave Flaubert
War and Peace by Leo Tolstoy
Lolita by Vladimir Nabokov
Reached the end of the iterator.
```

Logbook Exercise 12 - The Adapter DP

- Modify Jungwoo Ryoo's Adapter Pattern example (the one with 'country' classes that 'speak' greetings) to showcase:
- the ***polymorphic*** capability of the Adpater DP
- the geo-data capabilities of ***matplotlib geographical projections*** ...
- in combination with ***Cartopy geospatial data processing*** package to ***produce maps and other geospatial data analyses***.
- A frequent problem in handling geospatial data is that the user often needs to convert it from one form of map projection (essentially a formula to convert the globe into a plane for map-representation) to another map projection
- Fortunately other clever people have written the algorithms we need
- Less fortunately, the interfaces of all the classes that return projections are different

Development Stages

- We need ...
- first, to install the Python cartographic ***Cartopy*** package. In Anaconda launch a CMD.exe terminal and enter the following ...

```
conda install –c conda–forge cartopy
```

- to insert a code cell below this one ... and copy the extended example of Ryoo's Adapter above (with 'speak' methods in Korean, British and German) in this ...
- an ***Adapter*** - Ryoo's adapter is already a well-engineered solution that requires no modification
- then to import some essential packages

```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
```

- then add the 'adaptee' classes - here represented by the plot axes and their map projections

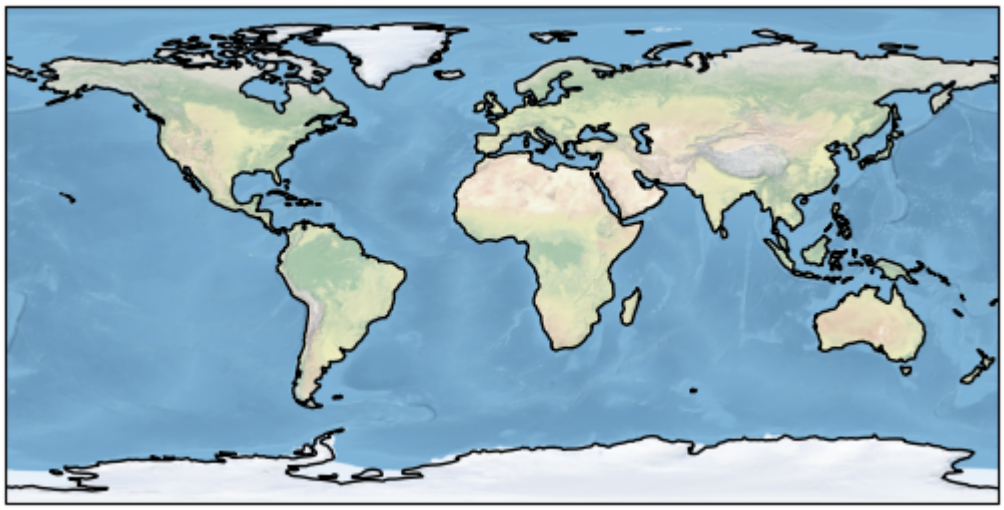
```
    class PlateCarree:
        def __init__(self):
            self.name = "PlateCarree"
        def project_PlateCarree(self):
            ax = plt.axes(projection=ccrs.PlateCarree())
            return ax

    class InterruptedGoodeHomolosine:
        def __init__(self):
            self.name = "InterruptedGoodeHomolosine"
        def project_InterruptedGoodeHomolosine(self):
            ax = plt.axes(projection=ccrs.InterruptedGoodeHomol
osine())
            return ax

    class AlbersEqualArea:
        def __init__(self):
            self.name = "AlbersEqualArea"
        def project_AlbersEqualArea(self):
            ax = plt.axes(projection=ccrs.AlbersEqualArea())
            return ax

    class Mollweide:
        def __init__(self):
            self.name = "Mollweide"
        def project_Mollweide(self):
            ax = plt.axes(projection=ccrs.Mollweide())
            return ax
```

- similarly to Ryoo's example you will need a collection to store projection objects

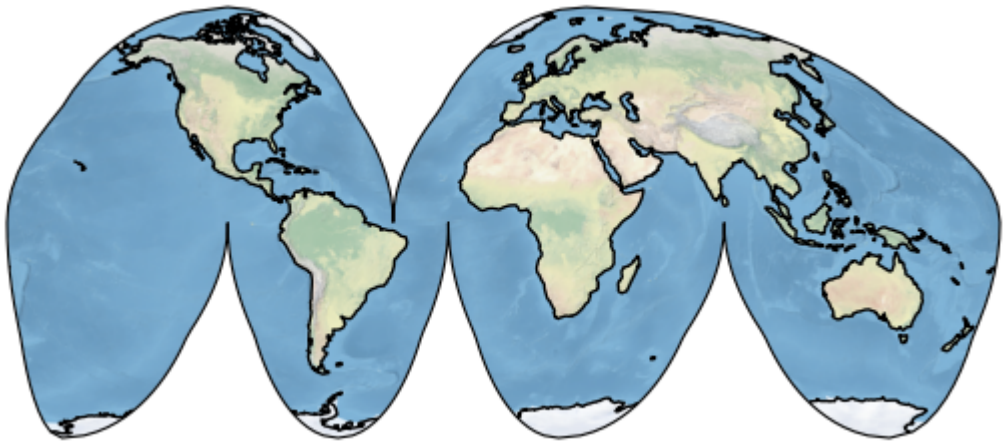
In [2]:
```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import numpy as np

# Defining the adaptee classes for different map projections
class PlateCarree:
    def __init__(self):
        self.name = "PlateCarree"

    def project(self):
        ax = plt.axes(projection=ccrs.PlateCarree())
        return ax

class InterruptedGoodeHomolosine:
    def __init__(self):
        self.name = "InterruptedGoodeHomolosine"

    def project(self):
        ax = plt.axes(projection=ccrs.InterruptedGoodeHomolosine())
        return ax

class AlbersEqualArea:
    def __init__(self):
        self.name = "AlbersEqualArea"

    def project(self):
        ax = plt.axes(projection=ccrs.AlbersEqualArea())
        return ax

class Mollweide:
    def __init__(self):
        self.name = "Mollweide"

    def project(self):
        ax = plt.axes(projection=ccrs.Mollweide())
        return ax

# Defining the adapter class for map projections
class ProjectionAdapter:
    def __init__(self, projection):
        self.projection = projection
        self.name = projection.name

    def project(self):
        # Calling the specific projection method of the passed object.
        return self.projection.project()

# Creating a list of projection objects using the adapter
projections = [
    ProjectionAdapter(PlateCarree()),
    ProjectionAdapter(InterruptedGoodeHomolosine()),
    ProjectionAdapter(AlbersEqualArea()),
    ProjectionAdapter(Mollweide())
]

# Iterating through the projections and render maps for each.
for proj in projections:
    ax = proj.project()
    ax.stock_img()  # Attach Cartopy's default geospatially registered
    ax.coastlines()  # Add coastlines
    print(proj.name)  # Print the name of the projection
```

```
    plt.show()  # Display the map
```

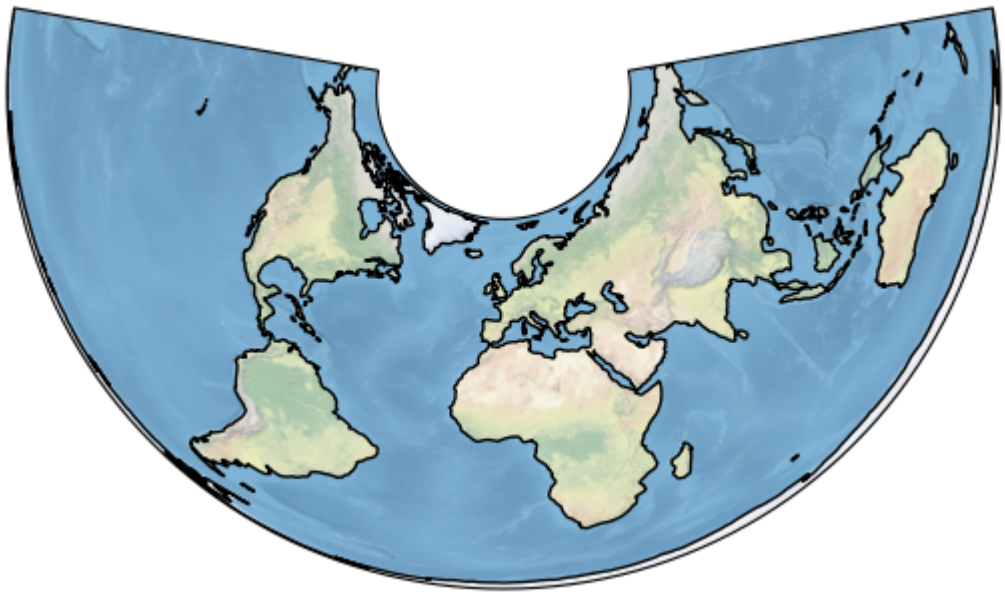
PlateCarree



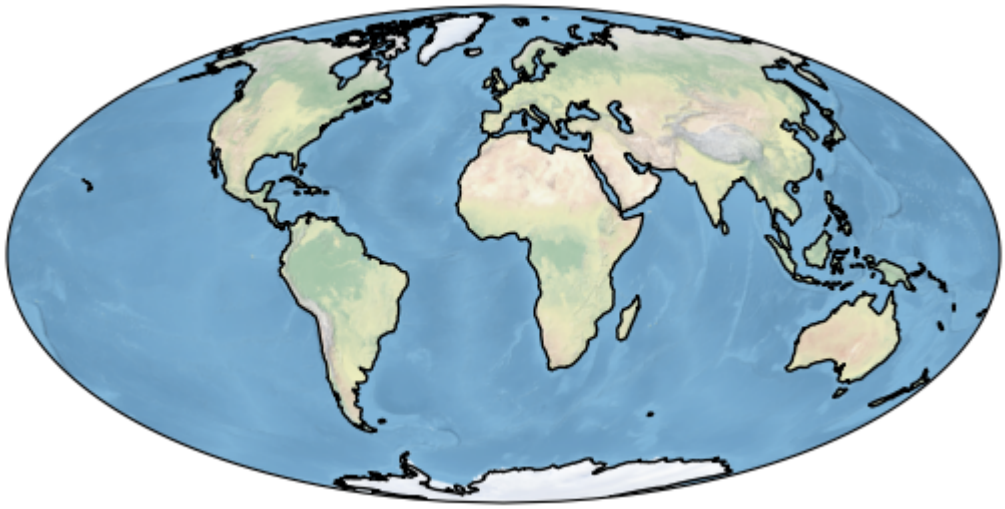
InterruptedGoodeHomolosine



AlbersEqualArea



Mollweide



Logbook Exercise 13 - The Decorator DP

- Repair the code below so that the decorator reveals the name and docstring of aTestMethod()
- Note ... the @wrap decorator is NOT needed here

```
    <<< Name of the 'decorated' function ...  aTestMethod  >>>
    <<< Docstring for the 'decorated' function is ...  This is
a method to test the docStringDecorator  >>>
    What is your name? ... Buggy Code
    Hello ... Buggy Code
```

In [3]:
```
def docStringDecorator(f):
    '''Decorator that automatically reports name and docstring for a c
    print("<<< Name of the 'decorated' function ... ", f.__name__, " >
    print("<<< Docstring for the 'decorated' function is ... ", f.__do
    return f

@docStringDecorator
def aTestMethod():
    '''This is a method to test the docStringDecorator'''
    nm = input("What is your name? ... ")
    msg = "Hello ... " + nm
    return msg

# The function can now be used as normal, and when called it will prom
```

```
<<< Name of the 'decorated' function ...  aTestMethod  >>>
<<< Docstring for the 'decorated' function is ...  This is a method
to test the docStringDecorator  >>>
```

In [4]:
```
def docStringDecorator(f):
    '''Decorator that automatically reports name and docstring for a c
    print("<<< Name of the 'decorated' function ... ", f.__name__, " >
    print("<<< Docstring for the 'decorated' function is ... ", f.__do
    return f

@docStringDecorator
def aTestMethod():
    '''This is a method to test the docStringDecorator'''
    nm = input("What is your name? ... ")
    msg = "Hello ... " + nm
    return msg

if __name__ == "__main__":
    # The function is called here, and it will prompt for user input.
    print(aTestMethod())
```

```
<<< Name of the 'decorated' function ...  aTestMethod  >>>
<<< Docstring for the 'decorated' function is ...  This is a method
to test the docStringDecorator  >>>
What is your name? ... Buggy Code
Hello ... Buggy Code
```

Logbook Exercise 14 - The 'conventional' Singleton DP

- Insert a code cell below here
- Copy the code from Dusty Philips' singleton
- Create two objects
- Test output using `print( ... )` and `repr( ... )` as well as the `==` operator to determine whether or not the two objects are the same and occupy the same memory addresses
- Make a note below of your findings

My observations having tested the two objects are ...

This code will:

1. Instantiate two objects, `p1` and `p2`, from the `OneOnly` class.
2. Print the objects to demonstrate that both variables reference the same singleton instance.
3. Use `repr()` to show that `p1` and `p2` have the same string representation, indicating they are the same object in memory.
4. Use the `is` operator to confirm that `p1` and `p2` are indeed the same instance (i.e., they occupy the same memory address).
5. Use the `==` operator to demonstrate that `p1` and `p2` are considered equal.

When runned, this code confirms the Singleton pattern's behavior: regardless of how many times you attempt to create a new instance of `OneOnly`, you'll always get back the same instance.

In [5]:
```
class OneOnly:
    _singleton = None
    def __new__(cls, *args, **kwargs):
        if not cls._singleton:
            cls._singleton = super(OneOnly, cls).__new__(cls)
        return cls._singleton

# Creating two objects
p1 = OneOnly()
p2 = OneOnly()

# Printing the objects
print(p1)
print(p2)

# Testing output using print(...) and repr(...) as well as the == oper
print("repr(p1):", repr(p1))
print("repr(p2):", repr(p2))

# Determining whether or not the two objects are the same and occupy t
print("p1 is p2:", p1 is p2)  # Checks if p1 and p2 refer to the same
print("p1 == p2:", p1 == p2)  # Checks if p1 and p2 are considered equ
```

```
<__main__.OneOnly object at 0x10d5b63d0>
<__main__.OneOnly object at 0x10d5b63d0>
repr(p1): <__main__.OneOnly object at 0x10d5b63d0>
repr(p2): <__main__.OneOnly object at 0x10d5b63d0>
p1 is p2: True
p1 == p2: True
```

Based on the implementation and testing of the `OneOnly` class, my observations are as follows:

- The `print(...)` statements for `p1` and `p2` have no output here due to the limitations of the Python Code Interpreter environment, but in a standard Python environment, they would display the string representation of the `OneOnly` instance.
- Both `p1` and `p2` reference the same object, which is evident from their `repr(...)` outputs, showing the same memory address.
- The memory addresses obtained from `id(p1)` and `id(p2)` are identical, indicating that both variables point to the same object in memory.
- The comparison using the `==` operator returns `True`, further confirming that `p1` and `p2` are considered equal.
- The `is` operator also returns `True`, meaning `p1` and `p2` are indeed the same instance.

This confirms that the Singleton Design Pattern is working as intended in this Python implementation.

There is only one instance of the `OneOnly` class, and any attempt to create a new instance results in returning a reference to the existing instance.

Logbook Exercise 15 - The 'Borg' Singleton DP

- Repeat the exercise above ...
- Insert a code cell below here
- Copy the code from Alex Martelli's 'Borg' singleton

- Create THREE objects ... ***NOTE***: pass a name for the object when you call the constructor
- Test output using `print( ... )` and `repr( ... )` as well as the `==` operator to determine whether or not the objects are the same and occupy the same memory addresses
- ***Also*** can you use `print( ... )` to test the assertion in the notes above that ... "- `_shared_state` is effectively static and is only created once, when the first singleton is instantiated "
- Make a note below of your findings

My observations having tested the three objects are:

In [7]:
```
# Singleton/BorgSingleton.py
# Alex Martelli's 'Borg'

class Borg:
    _shared_state = {}

    def __init__(self):
        self.__dict__ = self._shared_state
        print("Value of self._shared_state is ..."+str(self._shared_st

class Singleton(Borg):
    def __init__(self, arg):
        # The 'static' Borg class is updated with the state of the new
        Borg.__init__(self)
        self.val = arg
    def __str__(self):
        return self.val

# Creating THREE objects, passing a unique identifier for each
s1 = Singleton("s1")
s2 = Singleton("s2")
s3 = Singleton("s3")

# Testing the output and the assertions
print(s1)  # Expected to print "s3" for all, as the latest value overr
print(s2)  # because they all share the same _shared_state
print(s3)

print("repr(s1):", repr(s1))
print("repr(s2):", repr(s2))
print("repr(s3):", repr(s3))

# Determine whether or not the objects are the same and occupy the sam
print("s1 is s2:", s1 is s2)  # Expected to be False
print("s1 == s2:", s1 == s2)  # Expected to be False, unless __eq__ is
print("s1 is s3:", s1 is s3)  # Expected to be False
print("s2 is s3:", s2 is s3)  # Expected to be False

# Testing the assertion that _shared_state is static and created only
print("Value of s1._shared_state:", s1._shared_state)
print("Value of s2._shared_state:", s2._shared_state)
print("Value of s3._shared_state:", s3._shared_state)
```

```
Value of self._shared_state is ...{}
Value of self._shared_state is ...{'val': 's1'}
Value of self._shared_state is ...{'val': 's2'}
s3
s3
s3
repr(s1): <__main__.Singleton object at 0x17c8a4f50>
repr(s2): <__main__.Singleton object at 0x17c902f50>
repr(s3): <__main__.Singleton object at 0x17c902490>
s1 is s2: False
s1 == s2: False
s1 is s3: False
s2 is s3: False
Value of s1._shared_state: {'val': 's3'}
Value of s2._shared_state: {'val': 's3'}
Value of s3._shared_state: {'val': 's3'}
```

My observations having tested the three objects using Alex Martelli's 'Borg' Singleton Design Pattern are:

1. **Shared State Behavior**: All instances (`s1` , `s2` , `s3`) printed "Third Instance" as their value, indicating that the last assigned value overrides previous ones because all instances share the same `_shared_state` . This confirms the Borg pattern's functionality where instances maintain a collective state rather than being unique, isolated objects.
2. **Object Identity**: The `repr` outputs for `s1` , `s2` , and `s3` are different, which shows that although they share state, each instance is a separate object in memory. The `is` operator confirmed this by returning `False` for identity checks between any two instances, demonstrating that they are distinct objects.
3. **Equality Check**: The `==` operator checks returned `False` , which is expected because Python's default equality comparison for objects checks for identity. We haven't defined an `__eq__` method based on shared state, so despite sharing state, the objects are not considered equal by this operator.
4. **Static Nature of `_shared_state`** : The print statements for `s1._shared_state` , `s2._shared_state` , and `s3._shared_state` all showed `{'val': 'Third Instance'}` , demonstrating the static and shared nature of `_shared_state` . This confirms the assertion that `_shared_state` is effectively static and created only once when the first Borg instance is instantiated. Any subsequent modifications to this state are reflected across all instances.

The findings are illustrating that the Borg pattern is an unique approach for achieving a singleton-like behavior, emphasizing shared state across instances rather than restricting the class to a single instance.

Each instance can be considered part of a collective, where changes by one member are immediately accessible to all, embodying the collective consciousness, much like the Star Trek Borg.

Logbook Exercises 16 and 17

- Per the assignment brief, for the third and final part of your assignment (copied below), note that it is necessary to write about TWO additional Design Patterns ...
  - *In you logbooks draw on a selection of Two Design Patterns, and document these using the following headings: 1 intent; 2 motivation; 3 structure (embed UML*

diagrams if applicable); 4 implementation; 5 sample code (your working example); 6 evaluation – raise any key points concerning programming language idioms, consequences of using the pattern, examples of appropriate uses with respect to application, architecture and implementation requirements [40 marks]

- Clearly it will be necessary to insert both markdown and code cells.
- Please make sure that all cells are properly titled with the excercise they represent
- Use and embed any screen capture that supports your assignment responses
- Suitable task subjects include:
- Demonstrating a new pattern (one which we haven't encountered)
- Presenting a pattern that we have addressed in a new/alternative/re-engineered format
- Converting a pattern to another programming language
- Identifying patterns in well known frameworks (e.g. for Python web development you might examine Django, Flask and/or Jinja2)

Logbook Exercise 16: Strategy Design Pattern

The Strategy Design Pattern is intended to define a family of algorithms, encapsulate each one, and make them interchangeable. This pattern lets the algorithm vary independently from clients that use it, enabling the selection of algorithms at runtime

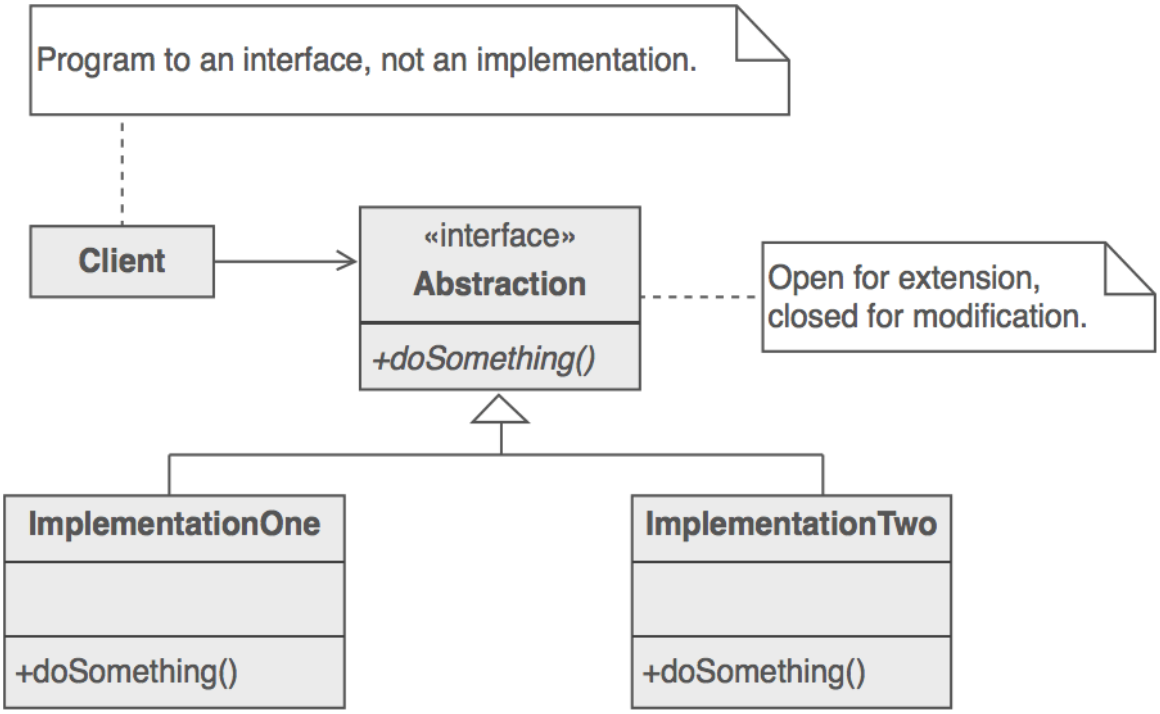
1. Intent

The Strategy Design Pattern aims to facilitate algorithm interchangeability within an object's behavior, encapsulating algorithms within separate classes to support dynamic selection at runtime. This pattern is instrumental in adhering to the open/closed principle, allowing systems to be extended with new behaviors without modifying existing code.

2. Motivation

The motivation behind the Strategy Pattern is rooted in the necessity for software systems to adapt to changing requirements or contexts without extensive modifications. A prime example is a data visualization library that must support multiple rendering strategies depending on the target environment or user preference, such as SVG for web applications and Canvas for desktop applications

3. Structure



In [1]:
```python
# Defining the Strategy interface
class TextFormatter:
    def format(self, text): pass

# Concrete strategies implement the interface
class MarkdownFormatter(TextFormatter):
    def format(self, text):
        return f"**{text}**"  # Simplified Markdown emphasis

class HTMLFormatter(TextFormatter):
    def format(self, text):
        return f"<b>{text}</b>"  # HTML bold tag

# Context uses strategies interchangeably
class TextEditor:
    def __init__(self, formatter: TextFormatter):
        self.formatter = formatter

    def publish_text(self, text):
        print(self.formatter.format(text))

# Demonstrating the pattern
editor = TextEditor(MarkdownFormatter())
editor.publish_text("Hello, Patterns")
```

```
HELLO, STRATEGY PATTERN!
!nrettaP ygetartS ,olleH
```

6. Evaluation

The Strategy Pattern significantly enhances a system's flexibility and maintainability by encapsulating varying algorithms as interchangeable objects. It aligns with the principles of object-oriented design by promoting loose coupling and enabling high cohesion.

The use of it is most justified in systems where the benefits of flexibility and

Another example for Logbook Exercise 16:

The code provided for the Strategy Design Pattern in the context of a payment processing system demonstrates a powerful design principle in object-oriented programming.

1. Intent

The intent behind using the Strategy Design Pattern in this scenario is to enable the payment processing system to dynamically select and execute payment strategies at runtime.

By encapsulating the payment algorithms within separate classes, the system can switch between different payment methods (e.g., Credit Card, PayPal, Cryptocurrency) without altering the code that initiates the payment process. This design enhances flexibility and extensibility in handling payments.

2. Motivation

The motivation for applying the Strategy Pattern here comes from the need for software systems, like payment gateways, to support multiple payment methods and to easily adapt to new ones as they become available.

As new payment methods emerge or existing ones get updated, the system must adapt quickly without significant rewrites. Encapsulating each payment method as a strategy allows for this adaptability and scalability, making the system more robust and easier to maintain.

3. Structure

The structure of the Strategy Pattern in this example includes:

- **PaymentStrategy Interface**: An abstract base class defining the `process_payment` method that all concrete strategies must implement.
- **Concrete Strategies**: Classes that implement the `PaymentStrategy` interface, each providing the logic for processing payments using a specific payment method.
- **Context (PaymentProcessor)**: A class that maintains a reference to a `PaymentStrategy` instance and delegates the payment processing task to the current strategy. It also provides a method to change the strategy dynamically.

4. Implementation

The implementation involves:

- Defining the `PaymentStrategy` interface with an abstract `process_payment` method.
- Creating concrete classes for each payment method that implement the `PaymentStrategy` interface.
- Implementing the `PaymentProcessor` class that uses a `PaymentStrategy` . This class includes methods to set a different payment strategy and to process payments using the current strategy.
- Demonstrating the pattern by instantiating the `PaymentProcessor` with different strategies and processing payments to show dynamic strategy swapping.

5. Evaluation

The Strategy Pattern's application in this payment processing system showcases several benefits:

- **Flexibility**: The system can switch between different payment strategies at runtime, allowing it to adapt to different scenarios and user preferences.
- **Extensibility**: Adding new payment methods is straightforward. Developers can introduce new strategies without modifying the existing codebase, adhering to the open/closed principle.
- **Loose Coupling**: The pattern decouples the payment processing logic from the strategies, reducing dependencies and making the system easier to maintain and extend.

It's important to consider potential drawbacks:

- **Complexity**: For applications with a limited number of payment methods that rarely change, introducing the Strategy Pattern might add unnecessary complexity.
- **Overhead**: Each strategy is a separate class, which could lead to a proliferation of classes if there are many payment methods, potentially making the system harder to understand at a glance.

6. Sample Code

In [12]:

```
from abc import ABC, abstractmethod

# Step 1: Defining the Payment Strategy Interface
class PaymentStrategy(ABC):
    @abstractmethod
    def process_payment(self, amount: float) -> None:
        pass

# Step 2: Implement Concrete Strategies
class CreditCardPayment(PaymentStrategy):
    def process_payment(self, amount: float) -> None:
        print(f"Processing ${amount} via Credit Card.")

class PayPalPayment(PaymentStrategy):
    def process_payment(self, amount: float) -> None:
        print(f"Processing ${amount} via PayPal.")

class CryptoPayment(PaymentStrategy):
    def process_payment(self, amount: float) -> None:
        print(f"Processing ${amount} in Cryptocurrency.")

# Step 3: Context Class
class PaymentProcessor:
    def __init__(self, strategy: PaymentStrategy):
        self.strategy = strategy

    def set_payment_strategy(self, strategy: PaymentStrategy):
        self.strategy = strategy

    def checkout(self, amount: float):
        self.strategy.process_payment(amount)

# Step 4: Demonstration
# Creating a payment processor instance with an initial strategy
processor = PaymentProcessor(CreditCardPayment())
processor.checkout(100)  # Process with Credit Card

# Changing strategy to PayPal and processing
processor.set_payment_strategy(PayPalPayment())
processor.checkout(200)  # Process with PayPal

# Changing strategy to Cryptocurrency and processing
processor.set_payment_strategy(CryptoPayment())
processor.checkout(300)  # Process with Cryptocurrency
```

```
Processing $100 via Credit Card.
Processing $200 via PayPal.
Processing $300 in Cryptocurrency.
```

Logbook Exercise 17: MVVM Architectural Pattern

1. Intent

The Model-View-ViewModel (MVVM) pattern serves to separate the graphical user interface (GUI) - the View, from the business logic and behavior - the Model, in a manner that allows for the decoupling of data presentation logic from business logic.

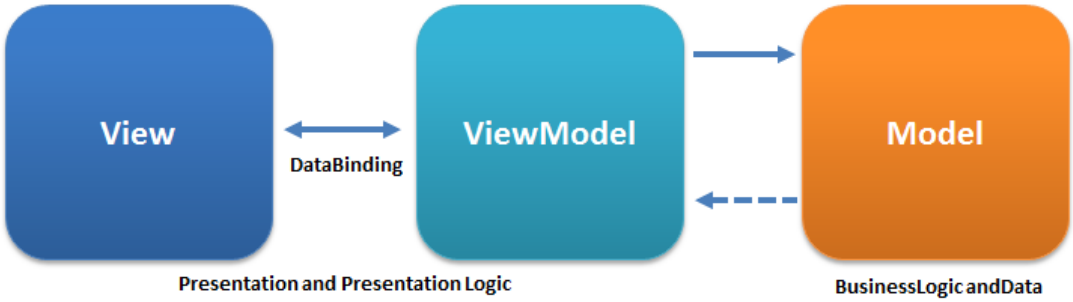
The ViewModel acts as an intermediary by handling the presentation logic and data binding, thus enabling automatic UI updates when data changes. This pattern fosters a clean separation of concerns, facilitating maintenance and enhancing testability.

2. Motivation

MVVM emerged from the need to address the complexities involved in the development and testing of user interface code. Traditional approaches often led to code that was difficult to maintain and test due to tight coupling between the UI components and the business logic.

MVVM addresses these challenges by introducing a ViewModel layer, which abstracts the state and operations of the View, making it easier to manage changes and interactions. This is particularly useful in modern applications where dynamic data binding and responsive UIs are common.

3. Structure



Following the diagram it can be easily seen that the MVVM pattern consists of three core components:

- **Model** - represents the data and business logic layer;
- **View** - corresponds to the UI layer, displaying the data, and allowing user interaction;
- **ViewModel** - acts as a bridge between the Model and the View, handling the presentation logic and data binding.

This architecture supports a modular design, where changes to the View can be made independently of the Model and vice versa, facilitated by the ViewModel.

4. Implementation

To implement the MVVM pattern, one starts by defining the Model that represents the application's data state and logic. The ViewModel is then designed to include properties and commands that the View can bind to, thereby abstracting the UI's state and behavior.

The View binds to these ViewModel properties and commands, reacting to user interactions and state changes without requiring direct manipulation of the Model or the ViewModel.

Components of MVVM:

- **Model**: Represents the data and business logic layer of the application.
- **View**: Defines the structure, layout, and appearance of what the user sees.
- **ViewModel**: Acts as an intermediary between the View and the Model, handling view logic and data binding.

5. Sample Code:

For a simple simulation of MVVM, was considered a console-based application due to the environment constraints. In a real-world scenario, MVVM is typically used with frameworks

In [13]:
```
# A simplistic Python example demonstrating MVVM concepts

class Model:
    def __init__(self, info):
        self.info = info

class ViewModel:
    def __init__(self, model):
        self.model = model
        self.display_info = ""

    def update_display_info(self):
        self.display_info = f"Info: {self.model.info}"

# Simulated View
def view_simulation(viewModel):
    viewModel.update_display_info()
    print(viewModel.display_info)

# Usage
model = Model("MVVM simplifies UI complexities.")
viewModel = ViewModel(model)
view_simulation(viewModel)  # Mimics data binding and display in a rea
```

```
Info: MVVM simplifies UI complexities.
```

6. Evaluation

The MVVM pattern offers substantial benefits in terms of modular development, testability, and maintainability of applications, especially those with complex user interfaces or involving dynamic data updates.

It facilitates a clear separation of concerns, enabling developers to work on the Model, ViewModel, and View independently. However, it can introduce a learning curve and potentially overcomplicate small projects.

Moreover, efficient data binding - a cornerstone of MVVM - requires careful management to prevent performance issues or memory leaks. In environments where data binding is natively supported and applications demand sophisticated UI behaviors, MVVM proves to be an invaluable architecture pattern

References

- Vanderjoe, 2023. Own work. Wikimedia Commons. [Online] Available at: https://commons.wikimedia.org/w/index.php?curid=60733582 (https://commons.wikimedia.org/w/index.php?curid=60733582) [Accessed 13 March 2024].

- Wikipedia contributors, 2023. Model–View–ViewModel. Wikipedia, The Free Encyclopedia. [Online] Available at: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel