DesignPatterns

October 23, 2023

1 CO664WBL Design Patterns

1.1 CW1 Design Patterns Logbook and Report

Jake Camacho - 22044195

GitHub link - https://github.com/Jakecamacho/DesignPattern.git

1.2 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',
This is my shopping tuple ('milk', 'eggs', 'bread', 'cheese', 'tea', 'coffee', 'rice', 'pasta'
This is my Shopping_set with duplicates removes {'rice', 'milk', 'pasta', 'cheese', 'eggs', 'te
This is my shopping_dict {'milk': '£1.20', 'eggs': '£0.87', 'bread': '£0.64', 'cheese': '£1.75
```

[ ]:
```
shopping_list = ["milk", "eggs", "bread", "cheese", "tea", "coffee", "rice",␣
 ↪"pasta", "milk", "tea"]

print(f" This is my shopping list {shopping_list}.")

shopping_tuple = ("milk", "eggs", "bread", "cheese", "tea", "coffee", "rice",␣
 ↪"pasta", "milk", "tea")
```

```
print(f" This is my shopping tuple {shopping_tuple}.")

shopping_set = set(shopping_list)
print(f" This is my shopping set with duplicated removed {shopping_set}.")

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(f" This is my shopping dict {shopping_dict}.")
```

1.3 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']
```

[ ]:
```
countries = ["USA", "Mexico", "Canada"]
countries.append("Greenland")
print(countries)

countries.remove("Greenland")
print(countries)

countries.insert(0, "Greenland")
print(countries)

sliced_countries = countries[:2]
print(sliced_countries)
```

```
sliced_countries = countries[-2]
print(sliced_countries)

sliced_countries = countries[1:3]
print(sliced_countries)
```

1.4 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])
```

[ ]:
```
a = [0,1,2,3,4,5,6,7,8,9,10]
b = [0,5,10,15,20,25]
topscores = { "Jo": 999, "Sue": 987, "Tara": 960, "Mike": 870}

print(f"List a is {a}")
print(f"List b is {b}")
print("The type of A is", type(a))

a = set(a)
b = set(b)
topscores = { "Jo": 999, "Sue": 987, "Tara": 960, "Mike": 870}

print(f"Set a is {a}")
print(f"Set b is {b}")
print("The type of A is", type(a))
print(f"Intersect of a and b is {a & b}")
print(f"Intersect of a and b is {a | b}")
```

```
print(f"Intersect of a and b is {a - b}")

dict_keys = topscores.keys()
print(f"Topscores dictionary keys are dict_keys {dict_keys}")
dict_values = topscores.values()
print(f"Topscores dictionary values are dict_values {dict_values}")
```

1.4.1 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

[ ]:
```
def address_machine(name, hs_number_street, town, postcode):
    address_format = "{0}, {1}, {2}, {3}".format(name, hs_number_street, town,␣
 ↪postcode)
    return address_format

name = ["T Cruise","D Francis","C White"]
hs_number_street = ["2 West St","65 Deadend Cls","15 Magdalen Rd"]
town = ["Canterbury", "Reading", "Oxford"]
postcode = ["CT8 23RD", "RG4 1FG", "OX4 3AS"]

new_list = [address_machine(name, hs_number_street, town, postcode) for name,␣
 ↪hs_number_street, town, postcode in zip(["T Cruise","D Francis","C White"],␣
 ↪["2 West St","65 Deadend Cls","15 Magdalen Rd"], ["Canterbury", "Reading",␣
 ↪"Oxford"], ["CT8 23RD", "RG4 1FG", "OX4 3AS"])]
print(new_list)

for items in new_list:
    print("\n" + items)
```

1.5 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
```

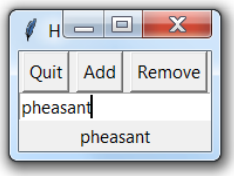
[ ]:
```
class Person:
    def __init__(self, first_name, second_name, postcode, age):
        self.fn = first_name
        self.ln = second_name
        self.pc = postcode
        self.age = age

    def greeting(self):
        print("Hello, my name is {0} {1}. I am {2} years old and my postcode is␣
 ↪{3}".format(self.fn, self.ln, self.age, self.pc))

class student(Person):
    def __init__ (self, first_name, second_name, postcode, age, degree_subject,␣
 ↪student_id):
        super().__init__(first_name, second_name, postcode, age)
        self.ds = degree_subject
        self.si = student_id
    def student_greeting(self):
        print("My student ID is {0} and I am studying {1}".format(self.si, self.
 ↪ds))

student1 = student("Dick", "Turpin", "HP11 2JZ", "32", "Highway Robbery",␣
 ↪"DT123456")
student1.greeting()
student1.student_greeting()
```

```
student2 = student("Dorothy", "Turpin", "S014 7AA", "32", "Law", "DT123457")
student2.greeting()
student2.student_greeting()

student3 = student("Oliver", "Cornwell", "0X35 14RE", "32", "History",␣
 ↪"0C123456")
student3.greeting()
student3.student_greeting()
```

1.6 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 higlighted below)

```
returned
['duck', 'duck', 'goose', 'penguin']
returned
['duck', 'duck', 'goose', 'penguin', 'chicken']
returned
['duck', 'duck', 'goose', 'penguin', 'chicken', 'turkey']
returned
['duck', 'duck', 'goose', 'penguin', 'chicken', 'turkey', 'pheasant']
returned
['duck', 'duck', 'goose', 'penguin', 'chicken', 'turkey']
```



[ ]:
```
#MVC_Template_01
#2014 May 23  by Steven Lipton http://makeAppPie.com
#Controller initializing MVC -- simplest version possible.
#Additional annotations by R Mather - Oct 2018

# MVC EXTENDED WITH DELETE BUTTON

# Tkinter is a Python binding to the Tk GUI toolkit/library - RM
from tkinter import *

# A Model-View-Controller framework for TKinter.
# Model: Data Structure. Controller can send messages to it, and model can␣
 ↪respond to message.
# View : User interface elements. Controller can send messages to it. View can␣
 ↪call methods from Controller when an event happens.
```

```
# Controller: Ties View and Model together. turns UI responses into chages in␣
 ↪data.



#Controller: Ties View and Model together.
#       --Performs actions based on View events.
#       --Sends messages to Model and View and gets responses
#       --Has Delegates - RM NOTE: essential for communications between view␣
 ↪and model via control

class MyController():
    def __init__(self,parent):
        self.parent = parent
        # CRITICAL PART OF THE MVC
        # RM NOTES:
        # MyController passes a reference to itself (self) when it creates␣
 ↪MyModel and MyView objects
        # This allows the MyModel and MyView objects to create 'delegates' (see␣
 ↪'vc' = virtual controller?)
        # The vc objects can then call methods on the MyControl object, thereby␣
 ↪allowing the View to update the Model via the Controller
        self.model = MyModel(self)    # initializes the model
        self.view = MyView(self)  #initializes the view
        # Initialize objects in view
        self.view.setEntry_text('Add or Remove a Label') # a non cheat way to␣
 ↪do MVC with tkinter control variables
        self.view.setLabel_text('Ready')

    #event handlers
    def quitButtonPressed(self):
        self.parent.destroy()
    def addButtonPressed(self):
        self.view.setLabel_text(self.view.entry_text.get())
        # CRITICAL PART OF THE MVC
        # RM NOTES - In one operation:
        #[1] Get the text from the VIEW;
        #[2] Add it to the MODEL by appending it to the list
        self.model.addToList(self.view.entry_text.get())
    def removeButtonPressed(self):
        self.view.setLabel_text(self.view.entry_text.get())
        self.model.removeFromList(self.view.entry_text.get())

    def listChangedDelegate(self):
        #model internally changes and needs to signal a change
        print(self.model.getList())
```

```
#View : User interface elements.
#       --Controller can send messages to it.
#       --View can call methods from Controller vc when an event happens.
#       --NEVER communicates with Model.
#       --Has setters and getters to communicate with controller

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.
 ↪quitButtonPressed).grid(row = 0,column = 0)
        addButton = Button(self.frame,text = "Add", command = self.vc.
 ↪addButtonPressed).grid(row = 0, column = 1)
        removeButton = Button(self.frame,text = "Remove", command = self.vc.
 ↪removeButtonPressed).grid(row = 0, column = 2)
        entry = Entry(self.frame,textvariable = self.entry_text).grid(row = 1,␣
 ↪column = 0, columnspan = 3, sticky = EW)
        label = Label(self.frame,textvariable = self.label_text).grid(row = 2,␣
 ↪column = 0, columnspan = 3, sticky = EW)

    def getEntry_text(self):
    #returns a string of the entry text
        return self.entry_text.get()
    def setEntry_text(self,text):
    #sets the entry text given a string
        self.entry_text.set(text)
    def getLabel_text(self):
    #returns a string of the Label text
        return self.label_text.get()
    def setLabel_text(self,text):
    #sets the label text given a string
        self.label_text.set(text)

#Model: Data Structure.
#   --Controller can send messages to it, and model can respond to message.
#   --Uses delegates from vc to send messages to the Control of internal change
```

```
#   --NEVER communicates with View
#   --Has setters and getters to communicate with Controller

class MyModel():
    def __init__(self,vc):
        self.vc = vc
        self.myList = ['duck','duck','goose']
        self.count = 0
#Delegates-- Model would call this on internal change
    def listChanged(self):
        self.vc.listChangedDelegate()
#setters and getters
    def getList(self):
        return self.myList
    def initListWithList(self, aList):
        self.myList = aList
    def addToList(self,item):
        print("returned")
        myList = self.myList
        myList.append(item)
        self.myList=myList
        self.listChanged()
    def removeFromList(self,item):
        print("returned")
        myList = self.myList
        myList.remove(item)
        self.myList=myList
        self.listChanged()

def main():
    # Create a root window from the Tk GUI class
    root = Tk()
    frame = Frame(root,bg='#0555ff' )
    root.title('Hello Penguins')
    app = MyController(root)
    root.mainloop()

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

1.7 Logbook Exercise 6 UPDATED

1.8 Extending LogBook 6

The MVC design pattern ensures a application consists of a controller, a data model and a view, for the display of information. These should be seperated out into different objects as can be seen below. There are many benefits of using the MVC design pattern; one being it promotes clear seperation of concerns, meaning each components responsibility is specific. The overall benefit of

this is it makes your code clear, reuseable and easily testable. As with any design patterns, there are some negatives. The main disadvantage with it MVC is it ability to increase the complexity of your code as you are required to manage three components and in larger projects this can involve more files to maintain.

For the logbook 6 I decided to extend the code by adding a 'search' function. When analysing the code initally I thought to do a refactor, however I hit a bit of a blocker and only realised after implementing the search function that I could have used setters within python to assit me.

The implementation of the search function is a great addition to the code as it allows for the user to search through the list and see if they can locate an item they intended to find. This extensions provides an additional functionaility to the MVC design pattern and demonstrates my understanding. The process works similar to the adding and removing of items from the list, where the controlled initiates a response when the searchbutton is pressed and the model then performs a look-up within the list to see if it can locate the search input. Depending on whether the item can be found or not, will depends on the output.

It was also crucial to add an if statement to state if the list was empty, then let the user know they cannot removed an item, and an item will need to be added to the list. The reason for implementing this functionaility was due to the knowledge gained from the data structure and algorithms unit, which aided in the extension of logbook 6.

[ ]:
```
# Tkinter is a Python binding to the Tk GUI toolkit/library - RM
from tkinter import *

# A Model-View-Controller framework for TKinter.
# Model: Data Structure. Controller can send messages to it, and model can␣
 ↪respond to message.
# View : User interface elements. Controller can send messages to it. View can␣
 ↪call methods from Controller when an event happens.
# Controller: Ties View and Model together. turns UI responses into chages in␣
 ↪data.



#Controller: Ties View and Model together.
#       --Performs actions based on View events.
#       --Sends messages to Model and View and gets responses
#       --Has Delegates - RM NOTE: essential for communications between view␣
 ↪and model via control

class MyController():
    def __init__(self,parent):
        self.parent = parent
        self.model = MyModel(self)     # initializes the model
        self.view = MyView(self)  #initializes the view
        # Initialize objects in view
        self.view.setEntry_text('Add or Remove a Label') # a non cheat way to␣
 ↪do MVC with tkinter control variables
```

```
        self.view.setLabel_text('Ready')

    #event handlers
    def quitButtonPressed(self):
        self.parent.destroy()
    def addButtonPressed(self):
        self.view.setLabel_text(self.view.entry_text.get())
        self.model.addToList(self.view.entry_text.get())
    def removeButtonPressed(self):
        self.view.setLabel_text(self.view.entry_text.get())
        self.model.removeFromList(self.view.entry_text.get())

    def searchButtonPressed(self):
        self.view.setLabel_text(self.view.entry_text.get())
        self.model.listSearch(self.view.entry_text.get())

    def listChangedDelegate(self):
        #model internally changes and needs to signal a change
        print(self.model.getList())


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.
 ↪quitButtonPressed).grid(row = 0,column = 0)
        addButton = Button(self.frame,text = "Add", command = self.vc.
 ↪addButtonPressed).grid(row = 0, column = 1)
        removeButton = Button(self.frame,text = "Remove", command = self.vc.
 ↪removeButtonPressed).grid(row = 0, column = 2)
        searchButton = Button(self.frame,text = "Search", command = self.vc.
 ↪searchButtonPressed).grid(row = 0, column = 4)
        entry = Entry(self.frame,textvariable = self.entry_text).grid(row = 1,␣
 ↪column = 0, columnspan = 5, sticky = EW)
        label = Label(self.frame,textvariable = self.label_text).grid(row = 2,␣
 ↪column = 0, columnspan = 5, sticky = EW)
```

```
    def getEntry_text(self):
    #returns a string of the entry text
        return self.entry_text.get()
    def setEntry_text(self,text):
    #sets the entry text given a string
        self.entry_text.set(text)
    def getLabel_text(self):
    #returns a string of the Label text
        return self.label_text.get()
    def setLabel_text(self,text):
    #sets the label text given a string
        self.label_text.set(text)


class MyModel():
    def __init__(self,vc):
        self.vc = vc
        self.myList = ['duck','duck','goose']

#Delegates-- Model would call this on internal change
    def listChanged(self):
        self.vc.listChangedDelegate()
#setters and getters
    def getList(self):
        return self.myList
    def initListWithList(self, aList):
        self.myList = aList
    def addToList(self,item):
        print(item + " has been added to your list!")
        myList = self.myList
        myList.append(item)
        self.myList=myList
        self.listChanged()
    def removeFromList(self,item):
        myList = self.myList
        myList.remove(item)
        self.myList=myList
        self.listChanged()
        if myList == []:
            print("Your list is currently empty, please add to your list!")
        elif item != myList:
            print ("The item you tried to remove cannot be found!")
        else:
            print(item + " has been removed from your list!")

    def listSearch(self, search):
```

```
        items = [item for item in self.myList if search in item]
        print("Perfct, it is a match!:", items)
        if search not in items:
            print ("oh no we cannot seem to find the result in your list!")
        self.listChanged()


def main():
    # Create a root window from the Tk GUI class
    root = Tk()
    root.title('List Management')
    app = MyController(root)
    root.mainloop()

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

1.9 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 concre
        # 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 slice
        fig1, ax1 = plt.subplots()
        ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%', shadow=True, startang
        ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
        plt.####()
```

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


# Nicely abstracted structure by which any model can notify any observer (view)␣
  ↪of changes in the model
class Subject(object): #Represents what is being 'observed'
```

```
    def __init__(self):
        self._observers = [] # This where references to all the observers are␣
 ↪being kept
                             # Note that this is a one-to-many relationship:␣
 ↪there will be one subject to be observed by multiple _observers

    def attach(self, observer):
        if observer not in self._observers: #If the observer is not already in␣
 ↪the observers list
            self._observers.append(observer) # append the observer to the list

    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 the list
            if modifier != observer: # Don't notify the observer who is␣
 ↪actually doing the updating
                observer.update(self) # Alert the observers!

# Represents the 'data' for which changes will produce notifications to any␣
 ↪registered view/observer objects
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', 'Lisp']
        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 the␣
 ↪labels

    @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 the data


# This is the 'standard' view/observer which also acts as an 'abstract' class␣
 ↪whereby deriving Bar/Chart/Table views override the update() method
# This 'abstracted' layer is always shown in examples but is important to␣
 ↪demonstrate potential polymorphic behaviour of update()
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 notify()␣
 ↪method in a concrete subject is invoked
        print("Generalised Viewer '{}' has: \nName = {}; \nLabels = {}; \nData␣
 ↪= {}".format(self._name, subject._name, subject._labels, subject._data))

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

    def update(self, subject): #Alert method that is invoked when the notify()␣
 ↪method in a concrete subject is invoked
        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 notify()␣
 ↪method in a concrete subject is invoked
        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()

class PieView(View):

    def update(self, subject): #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 = 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%%',␣
 ↪shadow=True, startangle=90)
        ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a␣
 ↪circle.
        plt.title("Programming Language Usage")
        plt.show()

#Let's create our subjects
m1 = Model("Model 1")
m2 = Model("Model 2") # This is never used!

#Let's create our observers
v1 = View("1: standard text viewer")
v2 = TableView("2: table viewer")
v3 = BarView("3: bar chart viewer")
v4 = PieView("4: pie chart viewer")
####

#Let's attach our observers to the first model
m1.attach(v1)
m1.attach(v2)
m1.attach(v3)
m1.attach(v4)
####

# Let's just call the notify() method to see all the charts in their unchanged␣
 ↪state
m1.notify()

# Now Let's change the properties of our first model
# Change 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]
```

1.10 Logbook Exercise 7 UPDATED

1.11 Refactoring LogBook 7

The observer pattern is typically used when the relationship between objects is one-to-many. This means the changing state of one object, updates all its dependents automatically. The code below follows the observer design pattern, demonstrating polymorphism through the use of the update method, encapsulation within the model classes and inheritence through the model class inherting from the subject class. The advantages of using this design pattern are it follows the open/closed principle. It also supports the idea of loose coupling between objects whom interact with each other and prevents the need for the change in the subject or observer classes, when data is sent to other objects. As with all the design systems it does also come with some disadvantages. One of the main disadvantages involving performance. If the observer is not correctly implemented, it can lead to inadvertent performance issues for they system, which is preferably avoided.

The design pattern overall can be seen as effective, allowing you to continuously add subscribers, without the existing functionaility being affected.

For the extension of logbook 7 I decided to introduce the super() as we were working with multiple inheritance. As a result of this it was suggested this would be a good function to use (W3Schools, 2023). In addition to this I adjusted the functionaility of the pieView class as the labels were overlapping. As can be seen above. In order to do this I added a legend to the piechart and rounded the statistics to 1 decimal place. This made it easier to visualise the piecharts and made them clearer.

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


# Nicely abstracted structure by which any model can notify any observer (view)␣
 ↪of changes in the model
class Subject(object): #Represents what is being 'observed'

    def __init__(self):
        self._observers = [] # This where references to all the observers are␣
 ↪being kept
                             # Note that this is a one-to-many relationship:␣
 ↪there will be one subject to be observed by multiple _observers

    def attach(self, observer):
        if observer not in self._observers: #If the observer is not already in␣
 ↪the observers list
            self._observers.append(observer) # append the observer to the list

    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 the list
            if modifier != observer: # Don't notify the observer who is␣
 ↪actually doing the updating
                observer.update(self) # Alert the observers!

# Represents the 'data' for which changes will produce notifications to any␣
 ↪registered view/observer objects
class Model(Subject): # Extends the Subject class

    def __init__(self, name):
        super().__init__()
        self._name = name # Set the name of the model
        self._labels = ['Python', 'C++', 'Java', 'Perl', 'Scala', 'Lisp']
        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 the␣
 ↪labels

    @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 the data


# This is the 'standard' view/observer which also acts as an 'abstract' class␣
 ↪whereby deriving Bar/Chart/Table views override the update() method
# This 'abstracted' layer is always shown in examples but is important to␣
 ↪demonstrate potential polymorphic behaviour of update()
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 notify()␣
 ↪method in a concrete subject is invoked
        print("Generalised Viewer '{}' has: \nName = {}; \nLabels = {}; \nData␣
 ↪= {}".format(self._name, subject._name, subject._labels, subject._data))

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

    def update(self, subject): #Alert method that is invoked when the notify()␣
 ↪method in a concrete subject is invoked
        super().update(subject)
        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 notify()␣
 ↪method in a concrete subject is invoked
        super().update(subject)
        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()

class PieView(View):

    def update(self, subject): #Alert method that is invoked when the notify()␣
 ↪method in a concrete subject is invoked
        super().update(subject)
        # Pie chart, where the slices will be ordered and plotted␣
 ↪counter-clockwise:
        labels = subject._labels
        sizes = subject._data
        Sum = sum(sizes)
        new_list = [ round(x/Sum * 100) for x in sizes]
```

```
        plt.pie(sizes, autopct='%1.1f%%', startangle=90)
        labels = ['{0} - {1:1.1f} %'.format(i,j) for i,j in zip(labels,␣
 ↪new_list)]
        plt.axis('equal')
        plt.legend(labels, loc='center right', bbox_to_anchor=(-0.1, 1.),
           fontsize=8)
        plt.title("Programming Language Usage")
        plt.show()

#Let's create our subjects
m1 = Model("Model 1")

#Let's create our observers
v1 = View("1: standard text viewer")
v2 = TableView("2: table viewer")
v3 = BarView("3: bar chart viewer")
v4 = PieView("4: pie chart viewer")
####

#Let's attach our observers to the first model
m1.attach(v1)
m1.attach(v2)
m1.attach(v3)
m1.attach(v4)
####

# Let's just call the notify() method to see all the charts in their unchanged␣
 ↪state
m1.notify()

# Now Let's change the properties of our first model
# Change 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]
```

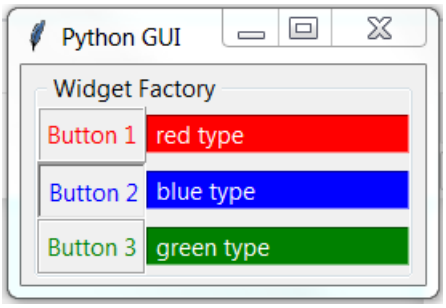
1.12 Logbook Exercise 8

- 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 Entry widget factory class - name it TextFactory()
- the TextFactory's Factory Method - name it createText(…)
- an abstract product - name it TextBase() and give it default attributes 'textvariable' and 'background'
- a getTextConfig(…) method

- 3x concrete text products - name these Text_1/2/3
- … and assign them textvariable values of " red/blue/green type" respectively
- … and assign them background values of 'red/blue/green' respectively
- to extend the OOP class with a createTextFields() method
- … that creates a factory object
- and the Entry fields … the code for the first Entry Field is as follows

```
# Entry field 1
sv=tk.StringVar()
tx = factory.createText(0).getTextConfig()[0]
sv.set(tx)
bg  = factory.createText(0).getTextConfig()[1]
action = tk.Entry(self.widgetFactory, textvariable=sv, background=bg, foreground="white"
action.grid(column=1, row=1)
```

- the end product should look something like this …



Challenge question

- At present the example has a 2x 'concrete' creators. As a comment just above the line `import tkinter as tk` briefly explain what you would need to do to refactor code so that the concrete creators extended an abstract class/interface called ***Creator*** (i.e. just as it appears in Gamma et al. (1995) and on slide 2)

[ ]:
```
'''
Example modified from Meier, B (2017) Python GUI Programming Cookbook, Second␣
 ↪Edition.  Packt Publishing

R Mather June 2020

'''

import tkinter as tk
from tkinter import ttk
from tkinter import Menu


class ButtonFactory():
    def createButton(self, type_):
        return buttonTypes[type_]()
```

```
class ButtonBase():
    relief     ='flat'
    foreground ='white'
    def getButtonConfig(self):
        return self.relief, self.foreground

class ButtonRidge(ButtonBase):
    relief     ='ridge'
    foreground ='red'

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

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

buttonTypes = [ButtonRidge, ButtonSunken, ButtonGroove]

class TextFactory():
    def createText(self, type_):
        return textTypes[type_]()

class TextBase():
    textVariable = 'white'
    background = 'white'
    def getTextConfig(self):
        return self.textVariable, self.background

class TextRed(TextBase):
    textVariable   ='red'
    background ='red'

class TextBlue(TextBase):
    textVariable ='blue'
    background ='blue'

class TextGreen(TextBase):
    textVariable ='green'
    background ='green'

textTypes = [TextRed, TextBlue, TextGreen]

class OOP():
    def __init__(self):
        self.win = tk.Tk()
```

```
        self.win.title("Python GUI")
        self.createWidgets()

    def createWidgets(self):
        self.widgetFactory = ttk.LabelFrame(text=' Widget Factory ')
        self.widgetFactory.grid(column=0, row=0, padx=8, pady=4)

        self.createButtons()
        self.createText()


    def createButtons(self):

        factory = ButtonFactory()

        # Button 1
        rel = factory.createButton(0).getButtonConfig()[0]
        fg  = factory.createButton(0).getButtonConfig()[1]
        action = tk.Button(self.widgetFactory, text="Button "+str(0+1),␣
↪relief=rel, foreground=fg)
        action.grid(column=0, row=1)

        # Button 2
        rel = factory.createButton(1).getButtonConfig()[0]
        fg  = factory.createButton(1).getButtonConfig()[1]
        action = tk.Button(self.widgetFactory, text="Button "+str(1+1),␣
↪relief=rel, foreground=fg)
        action.grid(column=0, row=2)

        # Button 3
        rel = factory.createButton(2).getButtonConfig()[0]
        fg  = factory.createButton(2).getButtonConfig()[1]
        action = tk.Button(self.widgetFactory, text="Button "+str(2+1),␣
↪relief=rel, foreground=fg)
        action.grid(column=0, row=3)

    def createText(self):
        factory = TextFactory()

        # Entry field 1
        sv=tk.StringVar()
        tx = factory.createText(0).getTextConfig()[0]
        sv.set(tx)
        bg  = factory.createText(0).getTextConfig()[1]
        action = tk.Entry(self.widgetFactory, textvariable=sv, background=bg,␣
↪foreground="white")
        action.grid(column=1, row=1)
```

```
        # Entry field 2
        sv=tk.StringVar()
        tx = factory.createText(1).getTextConfig()[0]
        sv.set(tx)
        bg  = factory.createText(1).getTextConfig()[1]
        action = tk.Entry(self.widgetFactory, textvariable=sv, background=bg,␣
 ↪foreground="white")
        action.grid(column=1, row=2)

        # Entry field 3
        sv=tk.StringVar()
        tx = factory.createText(2).getTextConfig()[0]
        sv.set(tx)
        bg  = factory.createText(2).getTextConfig()[1]
        action = tk.Entry(self.widgetFactory, textvariable=sv, background=bg,␣
 ↪foreground="white")
        action.grid(column=1, row=3)

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

1.13 Logbook Exercise 8 UPDATED

1.14 Refactoring LogBook 8

The factory design pattern is known as a creational design pattern. It is used when a superclass is present with numerous subclasses, and the return of the sub-class is dependent on the input. An example of this would be if I was a car broker and I need to the manufacturing details of different types of car, so what I would need to do is pass the name of the car I want and in return I would retrieve all the data about the car. There are many advantages with this design pattern with one of the main ones involving testing. By having different classes you are able to test the individual functionality of that class. As with all design patterns there are some downfalls and these involve having a large number of subclasses in larger applications which make it difficult to debug and navigate. If the file structure is maintained this issue can mitigated and does not oppose a problem.

1.15 Implementation

For the implementation I got a little confused and ended up getting stuck. I begun by creating an abstract factory and then passing this into my button and text factory. I think where I fell short was the re-organising of the OOP class where I needed to create a method which had the factory, button/text types and the create button methods. This would have provided me with the correct inputs to then proceed. As I was short for time I hadn't managed to combat this one but I believe with a little extra time I should have been able to complete it.

[ ]:
```
import tkinter as tk
from tkinter import ttk
from tkinter import Menu


class AbstractFactory():
    def creator(self, type_):
        pass

class ButtonFactory(AbstractFactory):
    def creator(self, type_):
        return buttonTypes[type_]()


class TextFactory():
    def createText(self, type_):
        return textTypes[type_]()


class ButtonBase():
    relief ='flat'
    foreground ='white'

    def relief(self):
        return self.relief

    def foreground(self):
        return self.foreground

class ButtonRidge(ButtonBase):
    relief     ='ridge'
    foreground ='red'

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

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

buttonTypes = [ButtonRidge, ButtonSunken, ButtonGroove]

class TextBase():
    textVariable = 'white'
    background = 'white'
```

```
    def textVariable(self):
        return self.text

    def background(self):
        return self.background

class TextRed(TextBase):
    textVariable ='red'
    background ='red'

class TextBlue(TextBase):
    textVariable ='blue'
    background ='blue'

class TextGreen(TextBase):
    textVariable ='green'
    background ='green'

textTypes = [TextRed, TextBlue, TextGreen]


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

    def createWidgets(self):
        self.widgetFactory = ttk.LabelFrame(text=' Widget Factory ')
        self.widgetFactory.grid(column=0, row=0, padx=8, pady=4)

        self.createButtons()
        self.createText()


    def createButtons(self):
        factory = ButtonFactory()
        rel = factory.createButton(0).getButtonConfig()[0]
        fg  = factory.createButton(0).getButtonConfig()[1]
        action = tk.Button(self.widgetFactory, text="Button "+str(0+1),␣
 ↪relief=rel, foreground=fg)
        action.grid(column=0, row=1)


    def createText(self):
        factory = TextFactory()
        sv=tk.StringVar()
```

```
        tx = factory.createText(0).getTextConfig()[0]
        sv.set(tx)
        bg  = factory.createText(0).getTextConfig()[1]
        action = tk.Entry(self.widgetFactory, textvariable=sv, background=bg,␣
 ↪foreground="white")
        action.grid(column=1, row=1)

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

1.16 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'!
Its food is 'Cat Food'!
```

[ ]:
```
class Animal:
    def speak(self):
        pass
    def __str__(self):
        pass

# This is the Abstract Factory
class AnimalFactory:
    def get_pet(self):
        pass
    def get_food(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 factory
class DogFactory(AnimalFactory):
    """Concrete Factory"""
    def get_pet(self):
        """Returns a Dog object - known as a concrete product"""
        return Dog()
    def get_food(self):
        """Returns a Dog Food object- known as a concrete product"""
        return "Dog Food!"

class Cat(Animal):
    def speak(self):
        """Returns a Dog object - known as a concrete product"""
        return "Meow!"
    def __str__(self):
        """Returns a Cat Food object- known as a concrete product"""
        return "Cat"

# Concrete factory
class CatFactory(AnimalFactory):
    def get_pet(self):
        return Cat()
    def get_food(self):

        return "Cat Food!"

#  The Client
class PetStore:
    # PetStore houses our Abstract Factory
    def __init__(self, pet_factory: AnimalFactory):
        # pet_factory is our Abstract Factory
        self._pet_factory = pet_factory
    def show_pet(self):
        #Utility method displays details of objects returned by DogFactory
        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))
```

```
# Create a Concrete Factory for Dog
dog_factory = DogFactory()
# Create a pet store housing the Dog Factory
dog_shop = PetStore(dog_factory)
# Invoke the utility method to show the details of our pet (Dog)
dog_shop.show_pet()

# Create a Concrete Factory for Cat
cat_factory = CatFactory()
# Create a pet store housing the Cat Factory
cat_shop = PetStore(cat_factory)
# Invoke the utility method to show the details of our pet (Cat)
cat_shop.show_pet()
```

1.17 Logbook Exercise 9 UPDATED

1.18 Refactoring LogBook 9

The abstract factory is classed as a creational design pattern. The design pattern provides a great way to encapsulate groups of factories and through using the abstract factory method we are able to produce objects of similar types. The pattern is very useful as it seperates the object creation details from their usage, ensuring the code is not dependent on previously defined objects. This is great as it ensures we wont have to alter these defined objects when we create a new object. The advantages is the design pattern follows the single responsibility principle which is where the interface code is seperate from the applications primary business logic. It is also very flexible and maintainable, with existing classes, whose functionailiy is no longer compatiable with the interfave, still being reused by the adaptors. Some of the disadvantages is the design pattern is not great for every situation when integrating compatible interfaces as it increases performance overhead so could just be easier to refactor the code base.

1.19 Implementation

For the extension I decided to use pythons built-in abstractmethod function (Abstract factory in Python, 2023). As you can see we have some Abstract classes below which contrain abstract methods. These are methods that are declared, containing no implementation. The abstract classes require subclasses to provide the implementation for the abstract methods and cannot be instantiated. The reason I used this is because it can make sure the classes follow the correct structure and implement all the abstract methods defined with the abstract class. The further benefit of using this function is you can set the types so when you call the client with one of the concrete factories this should provide you with their return statements.

[ ]:
```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self) -> str:
```

```
        pass

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

class Cat(Animal):
    def speak(self) -> str:
        """Returns a Dog object - known as a concrete product"""
        return "Meow!"
    def __str__(self) -> str:
        """Returns a Cat Food object- known as a concrete product"""
        return "Cat"

# This is the Abstract Factory
class AnimalFactory(ABC):
    @abstractmethod
    def get_pet(self) -> Animal:
        pass
    @abstractmethod
    def get_food(self) -> str:
        pass

# Concrete factory
class DogFactory(AnimalFactory):
    """Concrete Factory"""
    def get_pet(self) -> Animal:
        """Returns a Dog object - known as a concrete product"""
        return Dog()
    def get_food(self) -> str:
        """Returns a Dog Food object- known as a concrete product"""
        return "Dog Food!"


# Concrete factory
class CatFactory(AnimalFactory):
    def get_pet(self) -> Animal:
        return Cat()
    def get_food(self) -> str:
        return "Cat Food!"

#  The Client
class PetStore:
```

```
    """ PetStore houses our Abstract Factory """
    def __init__(self, pet_factory: AnimalFactory) -> None:
        """ pet_factory is our Abstract Factory """
        self._pet_factory = pet_factory
    def show_pet(self):
        """ Utility method displays details of objects returned by DogFactory␣
↪"""
        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))

for factory in (DogFactory(), CatFactory()):
    #Create a pet store housing our Abstract Factory
    shop = PetStore(factory)

    #Invoke the utility method to show the details of our pet
    shop.show_pet()
```

1.20 Logbook Exercise 10

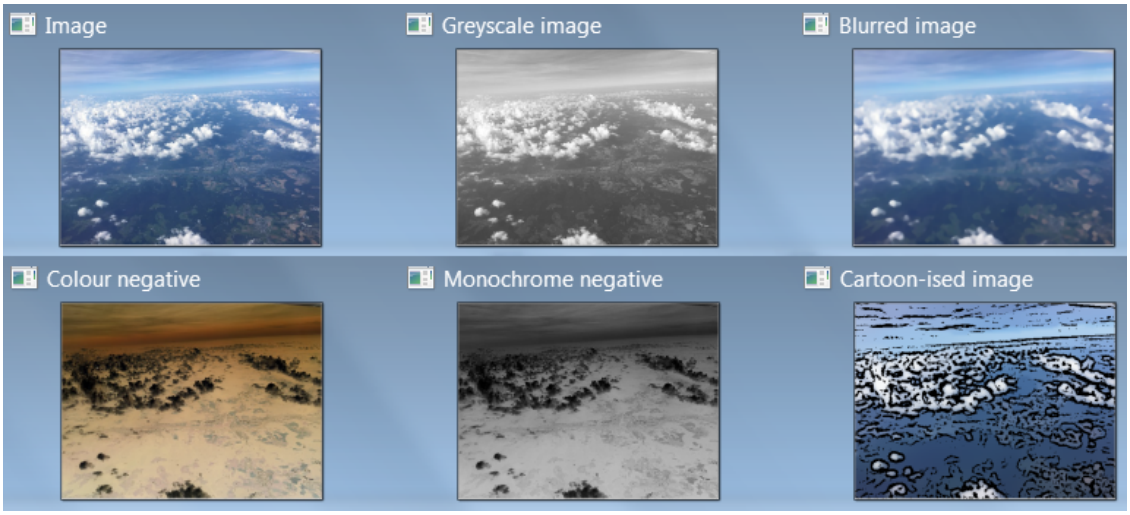
- Modify Jungwoo Ryoo's Strategy Pattern to showcase ***OpenCV*** capabilities with different image processing strategies
- We will use the ***OpenCV*** (Open Computer Vision) library which has been reproduced with Python bindings
- OpenCV has many standard computer science image-processing filters and 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 and https://www.tutorialspoint.com/cartooning-an-image-using-opencv-in-python

1.20.1 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.



1.20.2 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 - Display 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 applying one large filter
    for _ in range(num_bilateral):
        img_color = cv2.bilateralFilter(img_color, d=9, sigmaColor=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 and 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 and enhance edges
    img_edge = cv2.adaptiveThreshold(img_blur, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BIN
    # Combine color image with edge mask & display picture, convert back to color, bit-AND wit
    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(0); cv2.destroyAllWindows()
```

[ ]:
```
# Example from Jungwoo Ryoo (2015)
import cv2 as cv
import numpy as np
```

```
# The 'types' module is needed to support dynamic creation of new types. In␣
 ↪this case we dynamically create a new method type
import types #Import the types module


# Although 'concrete' this performs the role of the Strategy interface in the␣
 ↪GoF example
class Strategy:
    """The Strategy Pattern class"""

    def __init__(self, function=None):
        self.name = "Default Strategy"
        self.image = "./Image/clouds.jpg"

        #If a reference to a function is provided, replace the execute() method␣
 ↪with the given function
        if function:
            self.execute = types.MethodType(function, self)

    # Although 'concrete' this performs an equivalent role to the GoF␣
 ↪"AlgorithmInterface()"
    def execute(self): #This gets replaced by another version if another␣
 ↪strategy is provided.
        """The defaut method that prints the name of the strategy being used"""
        print("The image {} is used to execute Strategy 0 - Display image".
 ↪format(self.image))
        img_rgb = cv.imread(self.image)
        cv.imshow('Image', img_rgb)

# Replacement method 1 - this is a concrete implementation of␣
 ↪'AlgorithmInterface'
def strategy_one(self):
    img_rgb = cv.imread(self.image)
    img_gray = cv.cvtColor(img_rgb, cv.COLOR_RGB2GRAY)
    cv.imshow('Greyscale image', img_gray)

# Replacement method 2 - this is a concrete implementation of␣
 ↪'AlgorithmInterface'
def strategy_two(self):
    img_rgb = cv.imread(self.image)
    img_blur = cv.medianBlur(img_rgb, 7)
    cv.imshow('Blurred image', img_blur)

def strategy_three(self):
    img_rgb = cv.imread(self.image)
    for x in np.nditer(img_rgb, op_flags=['readwrite']):
```

```
        x[...] = (255 - x)
    cv.imshow('Colour negative', img_rgb)

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

def strategy_five(self):
    #Use bilateral filter for edge smoothing.
    num_down = 2 # number of downsampling steps
    num_bilateral = 7 # number of bilateral filtering steps
    img_rgb = cv.imread(self.image)
    # downsample image using Gaussian pyramid
    img_color = img_rgb
    for _ in range(num_down):
        img_color = cv.pyrDown(img_color)
    # repeatedly apply small bilateral filter instead of applying one large␣
 ↪filter
    for _ in range(num_bilateral):
        img_color = cv.bilateralFilter(img_color, d=9, sigmaColor=9,␣
 ↪sigmaSpace=7)
    # upsample image to original size
    for _ in range(num_down):
        img_color = cv.pyrUp(img_color)
    #Use median filter to reduce noise convert to grayscale and apply median␣
 ↪blur
    img_gray = cv.cvtColor(img_rgb, cv.COLOR_RGB2GRAY)
    img_blur = cv.medianBlur(img_gray, 7)
    #Use adaptive thresholding to create an edge mask detect and enhance edges
    img_edge = cv.adaptiveThreshold(img_blur, 255, cv.ADAPTIVE_THRESH_MEAN_C,␣
 ↪cv.THRESH_BINARY, blockSize=9, C=2)
    # Combine color image with edge mask & display picture, convert back to␣
 ↪color, bit-AND with color image
    img_edge = cv.cvtColor(img_edge, cv.COLOR_GRAY2RGB)
    img_cartoon = cv.bitwise_and(img_color, img_edge)
    # display
    cv.imshow("Cartoon-ised image", img_cartoon); cv.waitKey(0); cv.
 ↪destroyAllWindows()

# Let's create our default strategy
s0 = Strategy()
# Let's execute our default strategy
s0.execute()
```

```
# Let's create the first variation of our default strategy by providing a new␣
 ↪behavior
s1 = Strategy(strategy_one)
# Let's set its name
s1.name = "Strategy One"
# Let's execute the strategy
s1.execute()

s2 = Strategy(strategy_two)
s2.name = "Strategy Two"
s2.execute()

s3 = Strategy(strategy_three)
s3.name = "Strategy Three"
s3.execute()

s4= Strategy(strategy_four)
s4.name = "Strategy Four"
s4.execute()

s5 = Strategy(strategy_five)
s5.name = "Strategy Five"
s5.execute()
```

1.21 Logbook Exercise 10 UPDATED

1.22 The Strategy Pattern

The strategy design pattern is noted as a behavioural design pattern, turning a set of behaviours into object, so they become interchangeable within the original object. The main aim of this method is to allow the client to select between different algorithms in order to complete a specific task. An example of this in the working world wold be a credit card purchase. When you make a purchase the card you obtain contains all the information. Here the card would be the interface, while the type of card you have would be your implementation. The credit card itself is replaceable and is fully independent of another. The benefits of this design pattern is it is very easily extendable, making it useful for large applications. A disadvantage associated with this design is that you must know your different strategies and how these differ. Overall the design pattern is useful for large systems but may not be suitable for a smaller code base (Araujo and Easton, 1996).

1.23 Implementation

For the extension of this logbook I made some subtle refactoring changes. Within the Strategy class I passed in a name parameter and set it to default. This means unless I specify the namee of the method, then it will just call the startegy class. Once doing this I then appended this items into a list so I would be able to exxecute through them, reducing the amount of code to maintain and making it clearer and more concise.

```
[ ]: # Example from Jungwoo Ryoo (2015)
import cv2 as cv
import numpy as np

# The 'types' module is needed to support dynamic creation of new types. In␣
 ↪this case we dynamically create a new method type
import types #Import the types module


# Although 'concrete' this performs the role of the Strategy interface in the␣
 ↪GoF example
class Strategy:
    """The Strategy Pattern class"""

    def __init__(self, Name = "Default", function=None):
        self.name = "Default Strategy"
        self.image = "./Image/clouds.jpg"

        #If a reference to a function is provided, replace the execute() method␣
 ↪with the given function
        if function:
            self.execute = types.MethodType(function, self)

    # Although 'concrete' this performs an equivalent role to the GoF␣
 ↪"AlgorithmInterface()"
    def execute(self): #This gets replaced by another version if another␣
 ↪strategy is provided.
        """The defaut method that prints the name of the strategy being used"""
        print("The image {} is used to execute Strategy 0 - Display image".
 ↪format(self.image))
        img_rgb = cv.imread(self.image)
        cv.imshow('Image', img_rgb)

# Replacement method 1 - this is a concrete implementation of␣
 ↪'AlgorithmInterface'
def strategy_greyscale(self):
    img_rgb = cv.imread(self.image)
    img_gray = cv.cvtColor(img_rgb, cv.COLOR_RGB2GRAY)
    cv.imshow('Greyscale image', img_gray)

# Replacement method 2 - this is a concrete implementation of␣
 ↪'AlgorithmInterface'
def strategy_blur(self):
    img_rgb = cv.imread(self.image)
    img_blur = cv.medianBlur(img_rgb, 7)
    cv.imshow('Blurred image', img_blur)
```

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

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

def strategy_cartoon(self):
    #Use bilateral filter for edge smoothing.
    num_down = 2 # number of downsampling steps
    num_bilateral = 7 # number of bilateral filtering steps
    img_rgb = cv.imread(self.image)
    # downsample image using Gaussian pyramid
    img_color = img_rgb
    for _ in range(num_down):
        img_color = cv.pyrDown(img_color)
    # repeatedly apply small bilateral filter instead of applying one large␣
 ↪filter
    for _ in range(num_bilateral):
        img_color = cv.bilateralFilter(img_color, d=9, sigmaColor=9,␣
 ↪sigmaSpace=7)
    # upsample image to original size
    for _ in range(num_down):
        img_color = cv.pyrUp(img_color)
    #Use median filter to reduce noise convert to grayscale and apply median␣
 ↪blur
    img_gray = cv.cvtColor(img_rgb, cv.COLOR_RGB2GRAY)
    img_blur = cv.medianBlur(img_gray, 7)
    #Use adaptive thresholding to create an edge mask detect and enhance edges
    img_edge = cv.adaptiveThreshold(img_blur, 255, cv.ADAPTIVE_THRESH_MEAN_C,␣
 ↪cv.THRESH_BINARY, blockSize=9, C=2)
    # Combine color image with edge mask & display picture, convert back to␣
 ↪color, bit-AND with color image
    img_edge = cv.cvtColor(img_edge, cv.COLOR_GRAY2RGB)
    img_cartoon = cv.bitwise_and(img_color, img_edge)
    # display
    cv.imshow("Cartoon-ised image", img_cartoon); cv.waitKey(0); cv.
 ↪destroyAllWindows()
```

```
def add_list (list, item):
    list.append(item)
    return list

list = []
add_list(list, Strategy())
add_list(list,Strategy("Strategy GreyScale", strategy_greyscale))
add_list(list,Strategy("Strategy Blur",strategy_blur))
add_list(list,Strategy("Strategy Colour Negative",strategy_colour_negative))
add_list(list,Strategy("Strategy Grey Negative", strategy_grey_negative))
add_list(list,Strategy("Strategy Cartoon", strategy_cartoon))


for items in list:
    items.execute()
```

1.24 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 exception
    print( next(iter_mylist))
except Exception as e:
    print(e)
    print(sys.exc_info())
```

[ ]:
```
import sys

myList = ["Anna Karenina by Leo Tolstoy", "Madame Bovary by Gustave Flaubert",␣
 ↪"War and Peace by Leo Tolstoy", "Lolita by Vladimir Nabokov"]

class Books:
    def __init__(self, books):
        self.books = books

    def __iter__(self):
        self.b = -1
```

```
        return self

    def __next__(self):
        self.b += 1
        if len(self.books) > self.b:
            return self.books[self.b]
        else:
            raise StopIteration()

books = Books(myList)
iter_mylist = iter(books)


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

1.25 Logbook Exercise 11 UPDATED

1.26 The Iterator design pattern

The iterator design pattern is known as a behavioural pattern and is used to transverse through a group of ojects. The advantages of this design pattern are they reduce code duplication and gives flexibility when needs to transverse different collections. Some of the disadvantages is you cannot change a collection whilst it is being transversed with an iterator.

1.27 Implementation

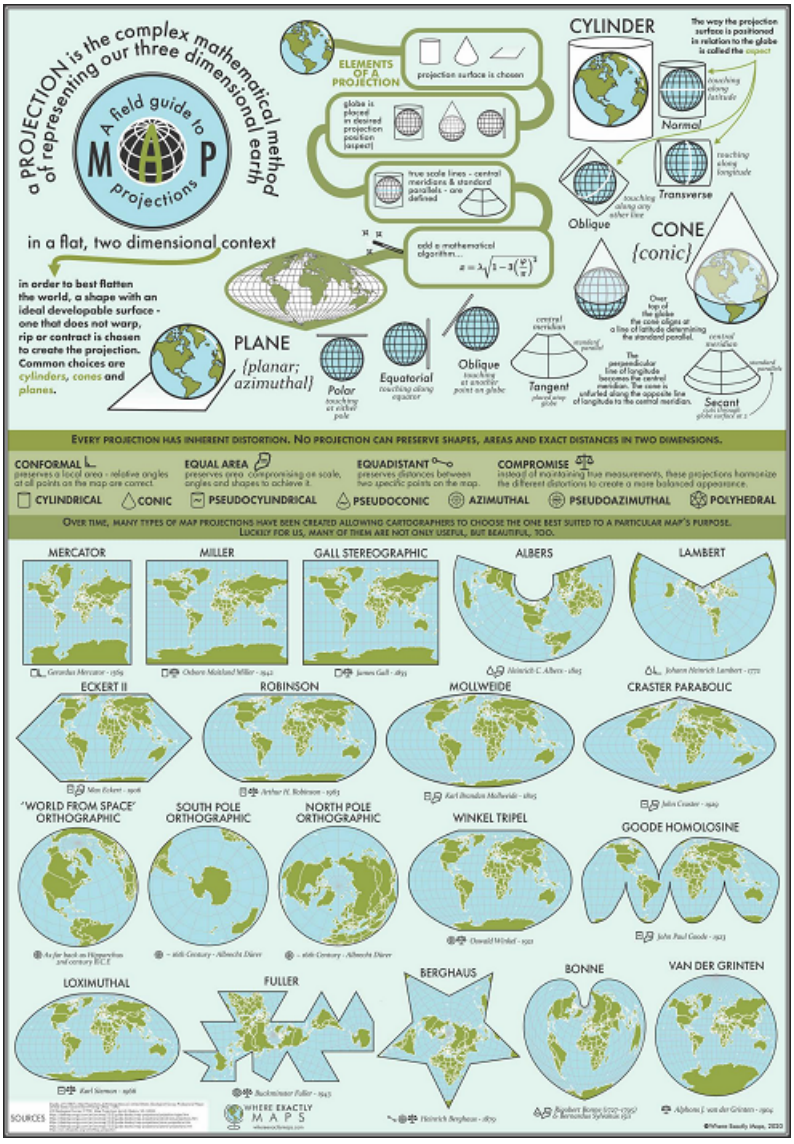
To extend this logbook I added in the next() function which works by returning the next item in an iterator (next function, 2023). This was perfect as I could use it within a while loop which would state while the condition was true, return the next element of the list. Once it has reched the end and the default value is not present then it would raise the stopIteration error.

[ ]:
```
import sys

myList = ["Anna Karenina by Leo Tolstoy", "Madame Bovary by Gustave Flaubert",␣
 ↪"War and Peace by Leo Tolstoy", "Lolita by Vladimir Nabokov"]

class Books:
    def __init__(self, books):
        self.books = books
```

```
    def __iter__(self):
        self.b = -1
        return self

    def __next__(self):
        self.b += 1
        if len(self.books) > self.b:
            return self.books[self.b]
        else:
            raise StopIteration()

books = Books(myList)
iter_mylist = iter(books)


try:
    while True:
        books = next(iter_mylist)
        print(books)
except Exception as e:
    print(e)
    print(sys.exc_info())
```

1.28 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

1.29 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
```

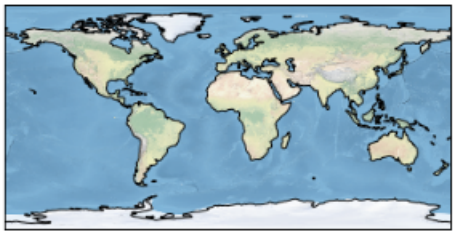
- then add the 'adaptee' classes - here represented by the plot axes and their map projections

```
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.InterruptedGoodeHomolosine())
        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
- again, simlarly to Ryoo, to create all the projection objects (e.g. `plateCarree =PlateCarree()`)
- again, similarly to Ryoo, to append to the collection key-value pairs for each projection and its projection method
- finally to traverse the list of objects to:

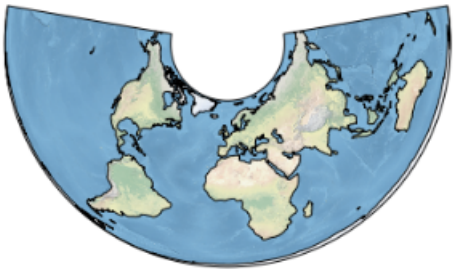
```
# Create an axes with the specified projection
ax = obj.project()
# Attach Cartopy's default geospatially registered map/image of the world
ax.stock_img()
# Add the coastlines - highlight these with a black vector
ax.coastlines()
# Print the name of the object/projection
print(obj.name)
# Plot the axes, projection and render the map-image to the projection
plt.show()
```

PlateCarree

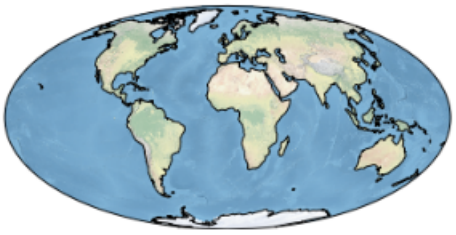


InterruptedGoodeHomolosine

AlbersEqualArea



Mollweide



- The output should look something like this …

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


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.InterruptedGoodeHomolosine())
                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

class Adapter:
    """This changes the generic method name to individualized method names"""

    # Here the ** operator allows a dictionary of key-value pairs to be passed␣
 ↪and resolved in function calls
    def __init__(self, object, **adapted_method):
        """Change the name of the method"""
        self._object = object

        # Add a new dictionary item that establishes the mapping between the␣
 ↪generic method name: speak() and the concrete method
        # For example, speak() will be translated to speak_korean() if the␣
 ↪mapping says so
        self.__dict__.update(adapted_method)

    def __getattr__(self, attr):
        """Simply return the rest of attributes!"""
        return getattr(self._object, attr)

obj = []

PlateCarree = PlateCarree()
InterruptedGoodeHomolosine = InterruptedGoodeHomolosine()
AlbersEqualArea = AlbersEqualArea()
Mollweide = Mollweide()

obj.append(Adapter(PlateCarree, project= PlateCarree.project_PlateCarree))
```

```
obj.append(Adapter(InterruptedGoodeHomolosine, project=␣
 ↪InterruptedGoodeHomolosine.project_InterruptedGoodeHomolosine))
obj.append(Adapter(AlbersEqualArea, project= AlbersEqualArea.
 ↪project_AlbersEqualArea))
obj.append(Adapter(Mollweide, project= Mollweide.project_Mollweide))

for objects in obj:
       # Create an axes with the specified projection
       ax = objects.project()
       # Attach Cartopy's default geospatially registered map/image of the␣
 ↪world
       ax.stock_img()
       # Add the coastlines - highlight these with a black vector
       ax.coastlines()
       # Print the name of the object/projection
       print(objects.name)
       # Plot the axes, projection and render the map-image to the projection
       plt.show()
```

1.30 Logbook Exercise 12 UPDATED

1.31 The Adaptor DP

The adapter design pattern is known as a structural design pattern, allowing the collaboration between objects within incompatible interfaces. A real-world analogy is when you go abroad and try to charge an electrical item but the charging ports are different and the only way around this is to use a travel adapter. The big advantage of this design pattern is it provides compatiable interfaces for the integration of legacy systems. Some of the downfalls with this design pattern is that you need to maintain the relationship between the adapter and adaptee class (Harrer et al, 2008).

1.32 Implementation

For this logbook I decided to extend an add in another class to show a different projection called Robinson. I have also decided to do a small refactor to make the code more clean cut and concise. The refactor will make it easier for the code to be updated and should result in less errors when updating

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


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.InterruptedGoodeHomolosine())
                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

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

class Adapter:

        # added a dictionary for the types we will recieve
        types = {
                        PlateCarree: "project_PlateCarree",
                        InterruptedGoodeHomolosine:␣
 ↪"project_InterruptedGoodeHomolosine",
                        AlbersEqualArea: "project_AlbersEqualArea",
                        Mollweide: "project_Mollweide",
                        Robinson: "project_Robinson"
        }

        def __init__(self, object):
                self._object = object
                self.project = getattr(object, self.types.get(type(object)))␣
 ↪#the adaptors method to get the object it wishes to adapt
```

```
        def __getattr__(self, attr):
                        return getattr(self._object, attr)

# placed the items in a list which will iniated the different class types
obj = [
           Adapter(PlateCarree()),
           Adapter(InterruptedGoodeHomolosine()),
           Adapter(AlbersEqualArea()),
           Adapter(Mollweide()),
           Adapter(Robinson())
]

for objects in obj:
        # Create an axes with the specified projection
        ax = objects.project()
        # Attach Cartopy's default geospatially registered map/image of the␣
 ↪world
        ax.stock_img()
        # Add the coastlines - highlight these with a black vector
        ax.coastlines()
        # Print the name of the object/projection
        print(objects.name)
        # Plot the axes, projection and render the map-image to the projection
        plt.show()
```

1.33 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 docStringDe
What is your name? ... Buggy Code
Hello ... Buggy Code
```

[ ]:
```
# REPAIR THE DECORATOR DP BELOW ...


def docStringDecorator(self):
    def decorator():
        '''Decorator that automatically reports name and docstring for a␣
 ↪decorated function'''
        print("<<< Name of the 'decorated' function ... ",self.__name__," >>> ")
        print("<<< Docstring for the 'decorated' function is ... ",self.
 ↪__doc__," >>>")
        ret = self()
        return ret
    return decorator
```

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

print(aTestMethod())
```

1.34 Logbook Exercise 13 UPDATED

1.35 The Decorator DP

The decorator design pattern is know to be able to make modifications to the behaviour of a function or class. By using the inbuilt wraps function in python you are able to wrap a function, extending the function of the function which has been wrapped, without making any permenant changes to the function itself. The benefits of this design pattern is you can expand a functions functionaility without the need of inheritance. As with any design pattern there is also some disadvantages. A key disadvantage involves debugging; as there in chain of decorated objects, this means that it could be difficult to spot errors within the code base.

1.36 Implementation

For the implementation I decided to use the wraps function. The wraps function does not affect the output directly within this example, but on expansion this would be a good method to use by updating the wrapper function, making it look like the wrapped function. In this example it would copy the **name** and **doc** attributes which is what was intended (Hussein and Mehanna, 2016). As a developer it is also crucial to added in test cases. One way to do this in python is by using assert (Okken, 2022). This would check to ensure the attributes we have wrapped, output as we would expect, showing the functionaility working correctly.

[ ]:
```
from functools import wraps

def docStringDecorator(function):
    # added in pythons internal wraps decorator
    @wraps(function)
    def decorator():
        '''Decorator that automatically reports name and docstring for a␣
 ↪decorated function'''
        print("<<< Name of the 'decorated' function ... ",function.__name__,"␣
 ↪>>> ")
        print("<<< Docstring for the 'decorated' function is ... ",function.
 ↪__doc__," >>>")
        return function()
    return decorator
```

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

print(aTestMethod())

# you can use assertion tests within your code to ensure your methods are␣
 ↪decorated with your decorator. If this is incorrect the code will raise and␣
 ↪assertion error.
# assert aTestMethod.__name__ != "aTestMethod"
```

1.37 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

1.37.1 My observations having tested the two objects are ...

The convential singleton is where all the code is contained within a class. The benefits of this is reduced memory usuage. By using this you can avoid allocating memory to objects of the same type, imporving the cache efficiency. A disadvantage associated with the convential singleton is the reduction of scalability

1.38 Observation

From my observations it could be conclude that Dusty Phillips singleton produces objects which are the same and occupy the same memory address. The reason for this is because it will only instantiate an object if there is not an object which currenlty exists. It uses the super() function which overrides the new instantiate. Below you can see when p1 class the oneOnly() class it creats an instance so when p2 is the called, it does not create a new instance and in fact returns the exisiting instance created.

[ ]:
```
## SINGLETON - Extended from Dusty Philips (2015) ##

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

```
p1 = OneOnly()
p2 = OneOnly()

print(repr(p1))
print(repr(p2))

print(p1 == p2)
```

1.39 Logbook Exercise 14

1.40 Lazy Singleton

With lazy initialisation the instance is only created upon when it is required. The creation is not done during the class loading process. This is great to use when the instance creation is very expensive as it will only do the creation when required, or if you only want to allocate resources when neeeded (Sioutis et al, 2019).

1.41 Implementation

Below I have implemented a lazy singleton. in the code below you can see when we initalise the singleton class and then try to print out its memory location we get None. The reason for this is because the object creation does not happen until you call the getInstance method. Once the getInstance method is called you can then see when we print out the memory location these are the same as the lazy singleton ensure only one instance is instiated and therfore will return the same memory location of the initalised object. This is how the lazy singleton achievex its name of being 'lazy' as it will only be initalised when required.

[ ]:
```
class Singleton:
    __instance = None
    def __init__(self):
        if not Singleton.__instance:
            print(" __init__ method called..")
        else:
            print("Instance created:", self.getInstance())
    @classmethod
    def getInstance(cls):
        if not cls.__instance:
            cls.__instance = Singleton()
        return Singleton.__instance


# class initialized, however the object will not be created.
s1 = Singleton()
s2 = Singleton()

# This currenrtly returns none type although the object has been initialized
print(s1._Singleton__instance)
```

```
print(s2._Singleton__instance)

# We now initialize the object by clling the getInstance method:
s1.getInstance()
s2.getInstance()

# as we not use
print(s1._Singleton__instance)
print(s2._Singleton__instance)


if s1 == s2:
   print ("we are the same object ")

else:
   print ("we are different objects")
```

1.42 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

1.42.1 My observations having tested the three objects are …

2 Singleton/BorgSingleton.py

3 Alex Martelli's 'Borg'

class Borg: _shared_state = dict()

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

class Singleton(Borg): def **init**(self, arg): # Here the 'static' Borg class is updated with the state of the new singleton object Borg.___init___(self) self.val = arg def **str**(self): return self.val

s1 = Singleton("s1") print(id(s1._shared_state)) s2 = Singleton("s2") print(id(s2._shared_state)) s3 = Singleton("s3") print(id(s3._shared_state))

print(repr(s1)) print(repr(s2)) print(repr(s3))

print (s1 == s2) print (s1 == s3) print (s2 == s3)

The benefit of using a borj single design patter over a singleton design pattern is due to the use of sub-classing. When you use sub-classing, the sub-classed object will have the same state as their parent objects. Therefore each sub-class will have its own state and therfore will produce different objects as can bee seen below.

3.1 Observation

By running tests it became visible that the shared_state is only created once on the instantiation of the first singleton. You can see this below as if you print the shared state of each of the singletons they all return the following 6079988416. This confirms the _shared_state is infact static. It was then noted to see if the objects are the same, and to see if whether they occupy the same memory location. From printing theses locations out and running a test, setting the objects equal to eachother, it was confirmed that the objects are infact different and also do not share the same memory locations. What this means is that they share the same state as their parent class, however each time a new singleton is instantiated it will be created at a different memory location.

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

class Borg:
    _shared_state = dict()

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

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

s1 = Singleton("s1")
print(id(s1._shared_state))
s2 = Singleton("s2")
print(id(s2._shared_state))
s3 = Singleton("s3")
print(id(s3._shared_state))


print(repr(s1))
print(repr(s2))
print(repr(s3))
```

```
print (s1 == s2)
print (s1 == s3)
print (s2 == s3)
```

3.2 Logbook Exercise 15

3.3 Thread-safe Singleton

Below I have implemented a thread-safe singleton. A thread-safe singleton is used in a multi-thread environment, ensuring the singleton is maintained. The 'instance is None' ensures multiple threads are not able to exist simultaneously. The benefit of this is by making a singleton thread-safe, multiple threads are able to access the instance, without concurrency issues being caused. This therefore improves the performance of the system. As with anything Muli-threads singletons have there disadvantages which include difficulty testing and debugging (Nesteruk, 2022). As it is a multi-thread environment it is harder to replicate the error and this makes testing difficult to reproduce.

Implementation:

Below I have implemented a thread safe singleton. From the code you can see the use of a 'with', which is where we recheck to see the presence of an instance. How this works is if we have two threads, both execute a singleton class at the same time. At this time the cls._instance is currently none and what this means is both of the threads will try to get the lock. One thread will get the lock while the other is blocked and in a waiting stage. The thread which acquires the lock will then exit the 'with' block, releasing the lock (cls._lock), which now allows for the second thread to be unblocked and acquires the lock. However because the cls._instance is None it will now fail. The second thread leaves execution and the instance already existing is now returned.

[ ]:
```
import threading

class ThreadSafeSingleton:
    _instance = {}
    _lock = threading.Lock()

    def __new__(cls, *args, **kwargs):
        # double-check locking where it rechecks the see if their is an instant␣
 ↪present
        if not cls._instance:
            with cls._lock:
                if not cls._instance:
                    instance = super().__new__(cls)
                    cls._instance[cls] = instance
        return cls._instance[cls]


def test_singleton(arg):
    singleton_instance = ThreadSafeSingleton(arg)
    print (singleton_instance)
```

```
    print("Memory address:", id(singleton_instance))


s1 = test_singleton("DEMO1")
s2 = test_singleton("DEMO2")
print (s1 == s2)
```

3.4 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)

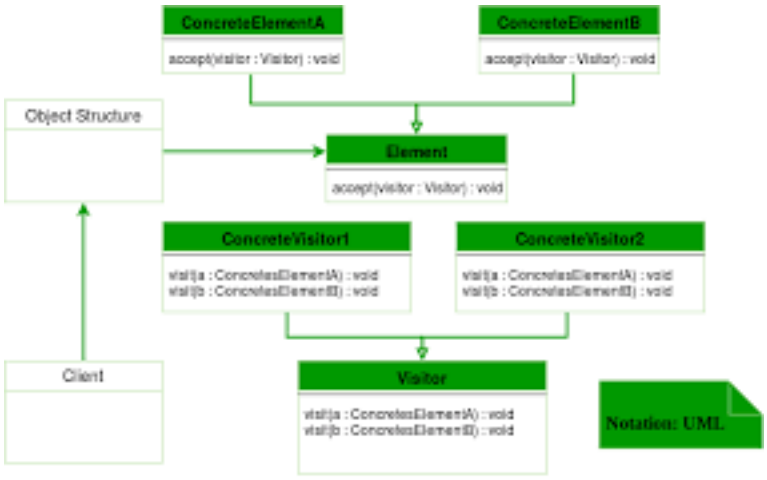
3.5 Logbook Exercise 16

3.6 Visitor Design Pattern

3.6.1 Intent

Visitor is known as a behavourial design pattern. The design pattern is used when you have a group of similar objects, which you need to preform an operation on. The pattern has two parts, one being a method known as visit() and this is called for each element within the data structure and is implemented by the vistor, and secondly an Accept() methods within visitable classes which accepts visitors. The design pattern is known to repersent an operation which is performed on the elements of an object, withough the class onwhich it operates on changing. An example of this is when an individual calls for a taxi, the taxi company works to dispath the cab to the customer. When entering the cab, the visitor, is no longer in control of their transportation and infact the taxi driver is.

3.7 Structure



3.7.1 Advantages

The key benefit of using a Visitor design patter is it is a well-functioning and pre-made way to extend entities which already exist in an object-oreintated system. If you require to add a new operation, this can simply be done by adding a new visitor. The design pattern is also known to allow all functional code to be centralised and the implementation of new operations only needs to be carried out in the visitor class, not needing to touch the remaining classes. One other benefit is that adding new items can be easy; all it requires is a change within the visitor interface, with the implementation and existing classes not being affected (Schordan, 2006).

3.7.2 Disadvantages

The big problem with the vistor design pattern is you should know the return types of the visit() methods at the time of design. Failure to do so could leave you having to change the interface and all the other implementations which can become a big hassle. Another key issues involved is having too many visitor interfaces as this can cause a problem when you wish to extend.

They key thing to note is this design pattern is the design pattern follows the deisgn principles of open-closed. What this states is classes should be open for extension but closed for modification (Palsberg and jay, 1998).

3.7.3 Implementation

This was probably one of the most difficult design patterns to get my head around, but after commiting to the cause I thought it would be good to delve into it a little more. For this implementation I took the example of imagining to run a shop and the numerous of different groceries a shop would have. Below we have a visitor object that has a visit method. Depending on the passed in argument will depend on how this method behaves as it is overloaded. The shop has a collection of elements which accepts a visitor and passses it to the each object element. Then groceries have an accept method, where it takes the visitor and visits it. I then used a dispatch decorator which allowed me to define two implementations with using the same function name, but when called they provide different outputs. From research it was noted this was good to use and it helped me to make my code more concise, consolidating multiple functions with similar behaviours into one.

[ ]:
```python
from abc import ABC, abstractmethod
from multipledispatch import dispatch

class Visitor(ABC):
    #Visitor interface

    @abstractmethod
    def visit(self):
        pass

class groceries(ABC):
    #interface of groceries

    @abstractmethod
    def display(self):
        pass

    @abstractmethod
    def accept(self, visitor: Visitor):
        pass

class groceries(groceries):
    #concrete parent class for groceries

    def __init__(self, title: str, cashier: str):
        self.title = title
        self.cashier = cashier

    def display(self):
        print(f"Display {self.title} by {self.cashier}")

    def accept(self, visitor: Visitor):
        visitor.visit(self)

class bread(groceries):
    #subclass for grocery
    def __init__(self, title: str, cashier: str):
        super().__init__(title,cashier)

class juice(groceries):
    #subclass for grocery"
    def __init__(self, title: str, cashier: str):
        super().__init__(title,cashier)


class shopVisitor(Visitor):
    #Concrete visitor
```

```
    @dispatch(bread)
    def visit(self, supplies):
        print(f"I will have one loaf of {supplies.title}, thanks {supplies.
 ↪cashier}")

    @dispatch(juice)
    def visit(self, supplies):
        print(f"I will have some {supplies.title} juice please, thanks␣
 ↪{supplies.cashier}")


class shop:
    #object structure
    def __init__(self):
        self.groceries = []

    def add_groceries(self, groceries: groceries):
        #adds groceries to its list
        self.groceries.append(groceries)

    def get_groceries(self):
        #get function of groceries list
        return self.groceries

    def accept(self, visitor: shopVisitor):
        #accepts a visitor and passes it to each grocery object
        for groceries in self.groceries:
            groceries.accept(visitor)

def run_a_shop():
    # create objects
    Bread = bread("bread", "Jonathan")
    Juice = juice("orange", "Jane")
    # creates the shop
    Shop= shop()
    Shop.add_groceries(Bread)
    Shop.add_groceries(Juice)
    # visit for the visitor
    Visitor = shopVisitor()
    Shop.accept(Visitor)


run_a_shop()
```

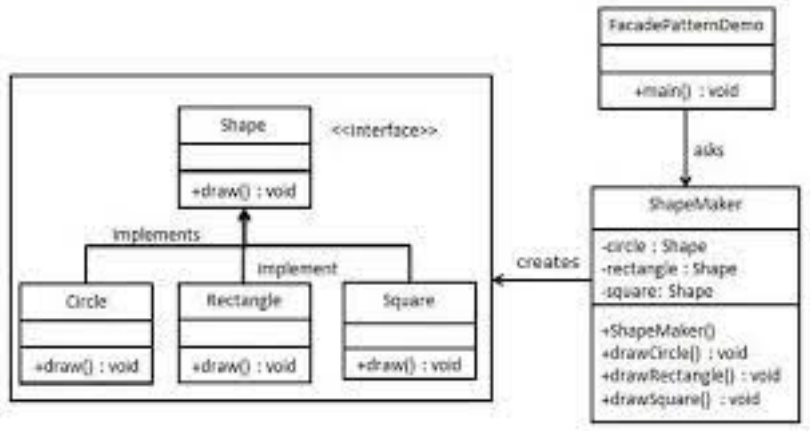
3.8 Logbook Exercise 17

3.9 Facade Design Pattern

3.9.1 Intent

Facade is known as a structural design pattern. It works by making the subsystem more easily useable by defining a high-livel interface. It therefore can be seen as a wrapping mechanism which, wraps these complicated subsystems, and from this creates a simple interface. A good example of facade is purchasing a new mobile phone. When a user contacts a phone provider they are greeted with a customer service repersenative which inturn acts as a facade. In order to fulfill your order, the customer repersenatice as to interact with many other services such as, billing, shipping, offers, etc… The main thing the facade does is it acts as a single point of entry for the user to be able to interact with the hidden services, diverging away of the complexity of how these services work (Alves and Bora, 2001).

3.9.2 Structure



3.9.3 Advantages

The Facade design pattern is usually used for complex systems, which when exposed to clients/users this is simplified. The purpose of facade is to hide complexity behind a userface which inturn makes it appear simpler on the outside. The benefits of the facade design pattern is your code becomes much more clearer and easier to read. When code becomes complex, the facade system is a great way to understand what the underlying code is doing. Making alterations to your code is also another key benefit. The way the code is structured allows you to make these changes at one given place, instead of needing to make changes throughout the code base, which can become complex and unmanageable. The facade design pattern also encourage decoupling (Bosch, 1996). It reduces direct dependencies between the components within the subsystem, making the system easier to test and more modular.

3.9.4 Disadvantages

Although the facade design pattern is known to make code clearer, it can also introduce unecessary complexity. It can be know to add additional layers of abstraction, making the code difficult to grasp and understand. Although simplicity is key, the facade pattern is known to hide how the

underlying subsystem works. Although this is great for users, when a bug arises, it can make it difficult for developers to debug and get to the root of the problem. Lastly, if the underlying system is known to change frequently, then it can prove to be a maintence burden in order to keed the facade in sync with subsystem.

The key thing to note when deciding whether to use the Facade system is the frequency of changes which will need to be made to the subsystem, the complexity of the underlying subsystem and lastly the potential impact it might have on performance and maintainability.

3.9.5 Implementation

Below I have implemented a Facade design pattern. From the code below you can see the car facade which displays which car has been selected and whether any of the parts of the car need replacing. From this you can see that the client would interact with the CarFacade which will then call different subsystems such as selectcar and replacement wheels, which will then go onto call further systems.

[ ]:
```
@staticmethod
class Audi:
    def drive(self, input):
        print(f" Driving around in my {input} today")

@staticmethod
class Vauxhall:
    def drive(self, input):
        print(f" Driving around in my {input} today")

@staticmethod
class Tesla:
    def drive(self, input):
        print(f" Driving around in my {input} today")

@staticmethod
class Skoda:
    def drive(self, input):
        print(f" Driving around in my {input} today")

@staticmethod
class Wheels:
    def drive(self, input):
        print(f" My wheels need replacing on my {input}")



# facade class which provides a unified interface for car selection
class carsFacade:
    def __init__(self):
        # Initialises the car instances for each supported input format
```

```
        self.Audi = Audi()
        self.Vauxhall = Vauxhall()
        self.Tesla = Tesla()
        self.Skoda = Skoda()

    def selectCar(self, input):
        # this method works by checking if the input supplied, returns a␣
 ↪specific output
        if input == "Audi":
            self.Audi.drive(input)
        elif input == "Vauxhall":
            self.Vauxhall.drive(input)
        elif input == "Tesla":
            self.Tesla.drive(input)
        elif input == "Skoda":
            self.Skoda.drive(input)
        else:
            print ("oh no, wrong input!")

    def replacementWheels(self, input):
        return Wheels().drive(input)



if __name__ == "__main__":
    # Creates a facade instance
    cars_facade = carsFacade()

    # uses the facade instance with car types
    cars_facade.selectCar("Audi")
    cars_facade.selectCar("Vauxhall")
    cars_facade.selectCar("Tesla")
    cars_facade.selectCar("Skoda")
    cars_facade.replacementWheels("Skoda")
```

References: - Abstract factory in Python (2023) Refactoring.Guru. Available at: https://refactoring.guru/design-patterns/abstract-factory/python/example (Accessed: 23 October 2023). - Alves, V. and Borba, P., 2001, October. Distributed adapters pattern: a design pattern for object-oriented distributed applications. In First Latin American Conference on Pattern Languages of Programming—SugarLoafPLoP, Rio de Janeiro, Brazil. - Araujo, L. and Easton, G., 1996. Strategy: where is the pattern?. Organization, 3(3), pp.361-383. - Bosch, J., 1996. Design Patterns as Language Constucts. - Harrer, A., Pinkwart, N., McLaren, B.M. and Scheuer, O., 2008. The Scalable Adapter design pattern: Enabling interoperability between educational software tools. IEEE Transactions on Learning Technologies, 1(2), pp.131-143. - Hussein, B. and Mehanna, A., 2016. A Design Pattern Approach to Improve the Structure and Implementation of the Decorator Design Pattern. Research Journal of Applied Sciences, Engineering and Technology, 13(5), pp.416-421. - Palsberg, J. and Jay, C.B., 1998, August. The essence of the visitor pattern. In Proceedings. The Twenty-Second Annual International Computer Software and Applications

Conference (Compsac'98)(Cat. No. 98CB 36241) (pp. 9-15). IEEE. - Schordan, M., 2006. The language of the visitor design pattern. J. Univers. Comput. Sci., 12(7), pp.849-867. - Sioutis, M., Paparrizou, A. and Condotta, J.F., 2019. Collective singleton-based consistency for qualitative constraint networks: Theory and practice. Theoretical Computer Science, 797, pp.17-41. - Nesteruk, D., 2022. Singleton. In Design Patterns in. NET 6: Reusable Approaches in C# and F# for Object-Oriented Software Design (pp. 113-130). Berkeley, CA: Apress. - Next Function (2023) Python next() function. Available at: https://www.w3schools.com/python/ref_func_next.asp (Accessed: 23 October 2023). - Okken, B., 2022. Python Testing with pytest. Pragmatic Bookshelf. - W3Schools (2023) Super Python Function , Python Super() function. Available at: https://www.w3schools.com/python/ref_func_super.asp#:~:text=Definition%20and%20Usage,that%20represent (Accessed: 23 October 2023).