

# Structural

## Structural Patterns

Focused on how objects and classes can be composed to form larger structures while keeping these structures flexible and efficient. These patterns deal with object composition and relationships between classes or objects. They help in building a flexible and reusable codebase by promoting better organization and modularity.

### Facade

According to Oxford Languages, a Facade is

“an outward appearance that is maintained to conceal a less pleasant or creditable reality.”

In the programming world, the "outward appearance" is the class or interface we interact with as programmers. And the "less pleasant reality" is the complexity that is hidden from us. So a Facade, is simply a wrapper class that can be used to abstract lower-level details that we don't want to worry about.

```
# Python arrays are dynamic by default, but this is an example of resizing.
```

```
class Array:
```

```
    def __init__(self):
```

```
        self.capacity = 2
```

```
        self.length = 0
```

```
        self.arr = [0] * 2 # Array of capacity = 2
```

```
# Insert n in the last position of the array
```

```
def pushback(self, n):
```

```
    if self.length == self.capacity:
```

```
        self.resize()
```

```
# insert at next empty position
```

```

self.arr[self.length] = n
self.length += 1

def resize(self):
    # Create new array of double capacity
    self.capacity = 2 * self.capacity
    newArr = [0] * self.capacity

    # Copy elements to newArr
    for i in range(self.length):
        newArr[i] = self.arr[i]
    self.arr = newArr

# Remove the last element in the array
def popback(self):
    if self.length > 0:
        self.length -= 1

```

Copy

## Adapter Pattern (Wrapper)

Adapter Pattern allow incompatible objects to be used together.

This supports *Composition over Inheritance* and *Open-Closed Principle*. Instead of modifying the base class, we extend the class behavior

Adapter is used when refactoring code, would never consider an adapter pattern from the start.

If a `MicroUsbCable` class is initially incompatible with `UsbPort`, we can create a adapter/wrapper class, which makes them compatible. In this case, a `MicroToUsbAdapter` makes them compatible, similar to how we use adapters in the real-world.

```

class UsbCable:
    def __init__(self):
        self.isPlugged = False

    def plugUsb(self):
        self.isPlugged = True

class UsbPort:

```

```
def __init__(self):
    self.portAvailable = True

def plug(self, usb):
    if self.portAvailable:
        usb.plugUsb()
        self.portAvailable = False

# UsbCables can plug directly into Usb ports
usbCable = UsbCable()
usbPort1 = UsbPort()
usbPort1.plug(usbCable)

class MicroUsbCable:
    def __init__(self):
        self.isPlugged = False

    def plugMicroUsb(self):
        self.isPlugged = True

class MicroToUsbAdapter(UsbCable):
    def __init__(self, microUsbCable):
        self.microUsbCable = microUsbCable
        self.microUsbCable.plugMicroUsb()

# can override UsbCable.plugUsb() if needed

# MicroUsbCables can plug into Usb ports via an adapter
microToUsbAdapter = MicroToUsbAdapter(MicroUsbCable())
usbPort2 = UsbPort()
usbPort2.plug(microToUsbAdapter)
```

---

Revision #5

Created 14 February 2024 22:54:48 by victor

Updated 1 April 2024 23:48:22 by victor