Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Exercise Sheet 3 – Solution

Software Architecture for Distributed Embedded Systems, WS 2021

Prof. Steinhorst, M. Sc. Regnath, M.Sc. Ernstberger

? Exercise 3.1: Patterns, Patterns, Patterns

1. Which categories of software patterns exists and what are the differences between them?

Solution:

- Structural: grouping and connecting objects using interfaces.
- Behavioral: specify communication and logic between objects.
- Creational: creation of instances of objects.

2. Name two patterns for each of the three categories.

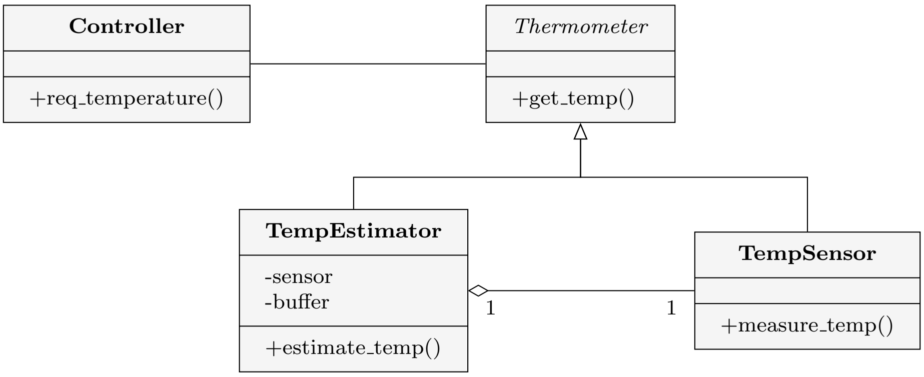
Solution:

- Structural: Adapter, Bridge, Proxy, Facade, ...
- Behavioral: Command, Observer, Memento, ...
- Creational: Singleton, Factory,

3. A message buffer is an often used pattern in communication. Which category of pattern would you assign it?

Solution: Behavioral pattern

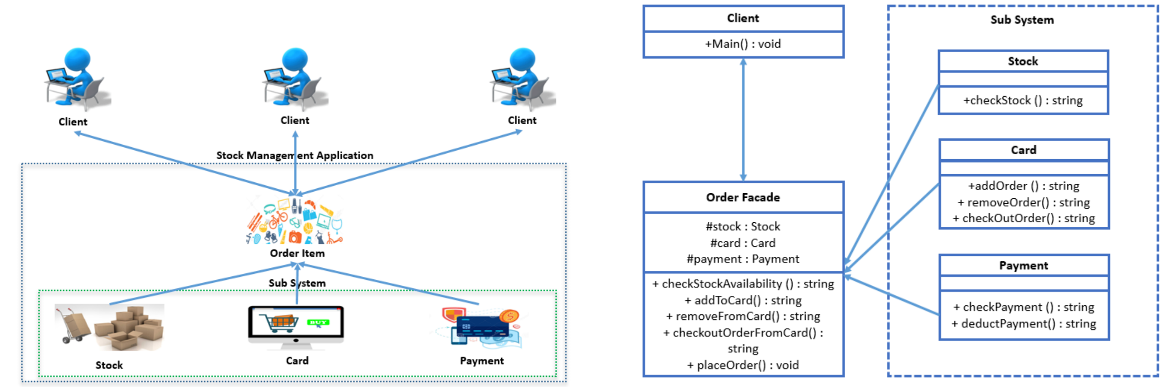
4. Which pattern is shown in the following UML diagram? What is the idea here?



Solution: The protective or remote Proxy Pattern. The idea is that temperature only changes slowly and does not need to be measured frequently. If the temperature was measured recently, the estimator will approximate the current temperature based on the last measurements.

5. When would you use the *Facade* pattern? Which problem does it solve?

Solution: If you have several different modules that need to be accessed in a specific sequence with specific parameters, you can use the Facade pattern to provide a simplified interface. It solves the problem of increased complexity if several modules need to be used together.



https://softwaredevelopmenttricky.blogspot.com/2017/12/facade-pattern.html

Exercise 3.2: Patterifying the Code

We want to improve the software of our Startup *AwesomeLights* from exercise 2.

1. Look at the following code. In this example our button wants to connect to a light using sockets. For security reasons, the connect command only accepts an IP address in a specific data format and not as string. Which design pattern would help here? Add your own code using the correct design pattern to make the connection work (see Moodle for full file).

```python
class IPv4_Address():
    def __init__(self, byte1, byte2, byte3, byte4):
        self.bytes = (byte1, byte2, byte3, byte4)

class Socket():

    def connect( self, ip_addr ):
        if isinstance(ip_addr, IPv4_Address): # check the correct format
            print("Socket:␣connecting␣to␣{}".format( str(ip_addr) ))
        else:
            print("Socket:␣Invalid␣IP␣format!")

light_ip = "192.168.1.42"

# connecting to the light
sock = Socket()
sock.connect( light_ip )  # THIS WILL FAIL
```

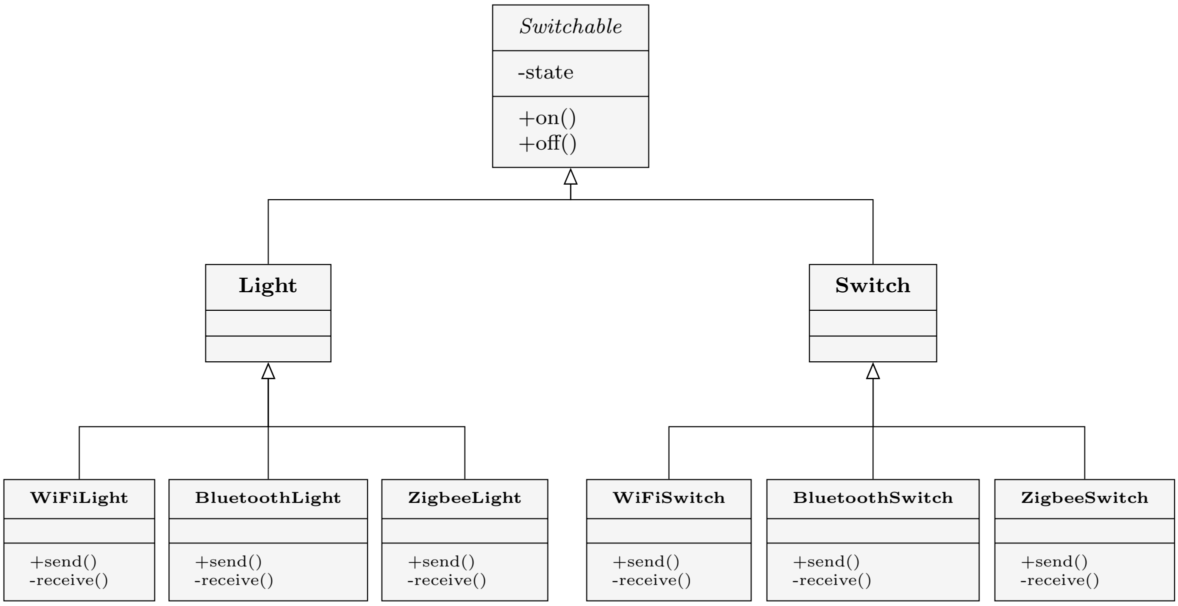
Solution:

Using Adaptor Pattern:

```python
class IPAdapter():
    def __init__(self, socket):
        self.socket = socket

    def connect( self, ip_addr ):
        try:
            a = ip_addr.split('.')
            ip_bytes = ( int(a[0]), int(a[1]), int(a[2]), int(a[3]) )
            if max(ip_bytes) > 255 or min(ip_bytes) < 0: raise ValueError
            ip_addr_struct = IPv4_Address(ip_bytes[0], ip_bytes[1],
    ↪ ip_bytes[2], ip_bytes[3] )
        except ValueError:
            print("Adapter:␣Invalid␣IP␣format!")
            return

        self.socket.connect(ip_addr_struct)

light_ip = "192.168.1.42"

# connecting to the light
sock = Socket()

ipadapter = IPAdapter( sock )
ipadapter.connect( light_ip )
```

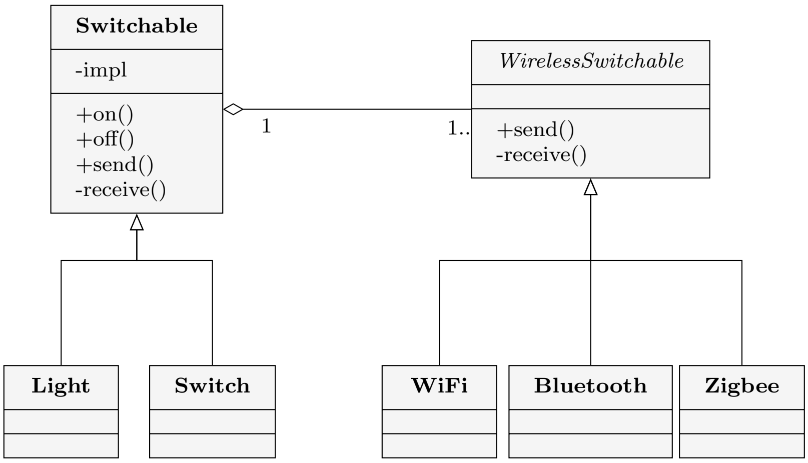
2. From a survey you have figured out that customers already have smart home solution that use Bluetooth or Zigbee instead of WiFi. You decide that it would be a good idea to offer switches and lights for each wireless standard (WiFi, Bluetooth, Zigbee). Extend the following UML diagram of the situation if you would need to support every combination individually, e.g. use classes such as `WiFiSwitch`, `ZigbeeLight`, etc that provide the functions `send()` and `receive()`.

Solution:

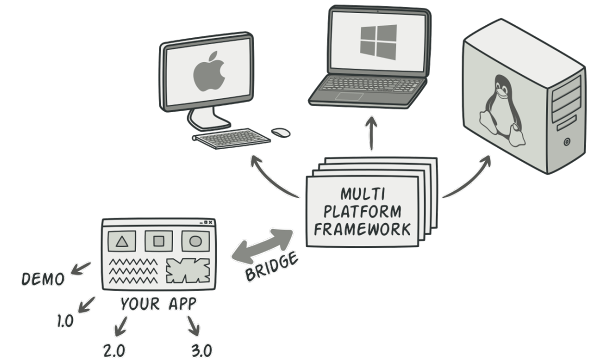


3. Redraw the UML using the bridge pattern. Make use of the class `Switchable` as well as `WirelessSwitchable`, which provide the functions `send()` and `receive()`.

Solution:



Note: In the embedded domain, you should in general consider the bridge pattern for a hardware abstraction layer if you need to support different target platforms.



https://refactoring.guru/design-patterns/bridge

4. Refactor the following code and implement the bridge pattern according to the UML diagram in the previous question (see Moodle for full file).

```
# Defining the Wireless Protocols
class WiFi():
    def sendTCP(self, msg):
    def recvTCP(self):

class Bluetooth():
    def request_service(self, srv):
    def handle_service(self):

class ZigBee():
    def sendIEEE(self, msg):
    def recvIEEE(self):


# Base Classes
class Switchable():
    def on(self):  raise NotImplementedError
    def off(self): raise NotImplementedError

# Derived Classes for all combinations
class WiFiSwitch(Switchable):
    def __init__(self, name):
        self.name = name
        self._wifi = WiFi()

    def on(self):
        self._wifi.sendTCP("ON")

    def off(self):
        self._wifi.sendTCP("OFF")

# ...
class WiFiLight(Switchable)
class BluetoothSwitch(Switchable)
class BluetoothLight(Switchable)
class ZigbeeSwitch(Switchable)
class ZigbeeLight(Switchable)
```

Solution:

Using a bridge between Switchable and WirelessSwitchable:

```
class WirelessSwitchable():
    def send(self, msg): raise NotImplementedError
    def receive(self):   raise NotImplementedError


class Switchable():
    def __init__(self):
        self._impl = None

    def set_protocol(self, protocol):
        self._impl = protocol

    def send(self, msg): self._impl.send(msg)
    def receive(self):   return self._impl.receive()

    def on(self):  raise NotImplementedError
    def off(self): raise NotImplementedError


class Switch(Switchable):
    def on(self):  self._impl.send("ON")
    def off(self): self._impl.send("OFF")


class Light(Switchable):
    def listen(self):
        packet = self._impl.receive()
        if packet == "ON":  self.on()
        if packet == "OFF": self.off()


# main
light = Light("Kitchen")
switch = Switch("Kitchen")

light.set_protocol( Bluetooth() )
switch.set_protocol( Bluetooth() )

switch.on()
light.listen()
```

5. What is the difference between Adapter and Bridge pattern?

Solution: Adapter connects unrelated components by translating interfaces Bridge lets different implementations of a base class vary independently from their interfaces.

For a bridge you could add a refined abstraction or an implementation without the need to change anything.