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

Exercise Sheet 4

Software Architecture for Distributed Embedded Systems, WS 2021/22

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

Exercise 4.1: Behavioral Patterns: Observer

1. We want to model the *AwesomeLights* scenario with the Observer pattern. Assuming the two classes `Switch`, `Light`, which class is the indepentent subject and which is the dependent observer?
2. Draw the UML diagram for the light and switch scenario using the observer pattern. Assume two lights and two switches. Use the classes `Observerable` and `Observer`.
3. Implement the observer pattern in the following code. Remove the class `Controller` and implement the classes `Observerable` and `Observer` (see Moodle for full file).

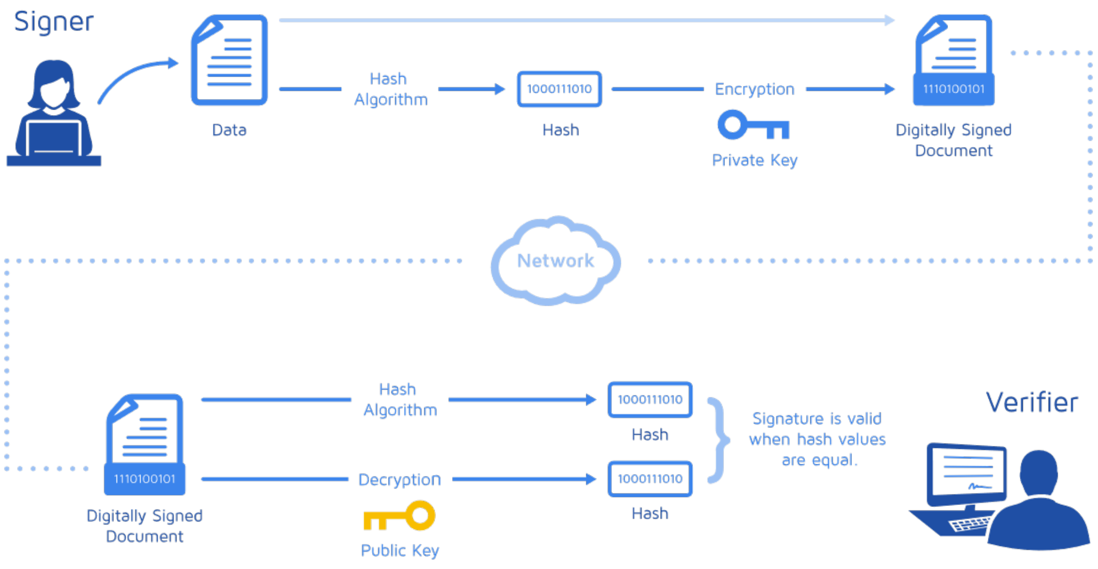
```python
class Switchable():
    def on(self):
    def off(self):

class Light(Switchable):

class Switch(Switchable):

class Controller(threading.Thread):
    def __init__(self, switches):
        self.switches = []+switches # expert question: why []+ ?
        self.light_map = [ [] ]*len(switches)

    def set_lights_for_switch(self, switch, lights):
        ...

    def run(self):
        while True:
            for switch_idx, lights in enumerate(self.light_map):
                if self.switches[switch_idx].was_switched():
                    for light in lights:
                        if light.is_on():
                            light.off()
                        else:
                            light.on()

# create lights
l1 = Light("Kitchen")
l2 = Light("Living␣Room")
l3 = Light("Bedroom")

# create buttons
s1 = Switch("Door")
s2 = Switch("Window")
s3 = Switch("Desk")


ctrl = Controller( [s1, s2, s3] )

ctrl.set_lights_for_switch( s1, [l1] )
ctrl.set_lights_for_switch( s2, [l1, l2] )
ctrl.set_lights_for_switch( s3, [l2, l3] )

ctrl.start()  # start thread
```

4. Is it possible to model the *AwesomeLights* scenario with the Model-View-Controller pattern? If yes, draw the UML diagram. If no, explain why it is impossible.

? Exercise 4.2: Behavioral Patterns: Strategy

From the ESI lecture "System Design for the Internet of Things" you have learned that security is often underestimated in IoT systems. You want to do it better and use Public Key Cryptography (PKC) in order to authenticate the messages between switches and ligths. Since you do not know which algorithm would best fit for a given setup, you decide to support `RSA`, `ECDSA`, and `XMSS` using the **strategy pattern**.

Signing the message works by calculating the hash of the message and then encrypting the hash using the secret key. Any receiver of the message can now "decrypt" the signature and compare it against the original message hash.



https://medium.com/@meruja/digital-signature-generation-75cc63b7e1b4

1. Draw an UML diagram for the three schemes `RSA`, `ECDSA`, and `XMSS` using the strategy pattern. Create the class `SignatureScheme` which provides the functions `set_algorithm(algo)`, `sign(msg)`, and `verify(msg, sig)` and the class `Algorithm` with the functions `encrypt(msg, secret_key)` and `decrypt(msg, public_key)`.

2. The following code provides cryptographic functions to authenticate messages with a signature. Improve the code with the strategy pattern to support all three algorithms in a unified way. Implement the class `Algorithm` that provides the functions `encrypt` and `decrypt` (see Moodle for full file).

```python
class RSA():
    def encrypt(self, msg, secret_key): ...
    def decrypt(self, msg, public_key): ...

class ECDSA(): ...
class XMSS():  ...

# main
msg = "hello␣world!"
secret_key = 42
public_key = 21

ecdsa = ECDSA()

# Sender: signing
msg_hash = calc_hash( msg )
sig = ecdsa.encrypt(msg_hash, secret_key)

# sending msg + sig to receiver ...

# Receiver: verifying
msg_hash = calc_hash( msg )
signed_hash = ecdsa.decrypt(sig, public_key)
is_valid = (signed_hash == msg_hash)
```