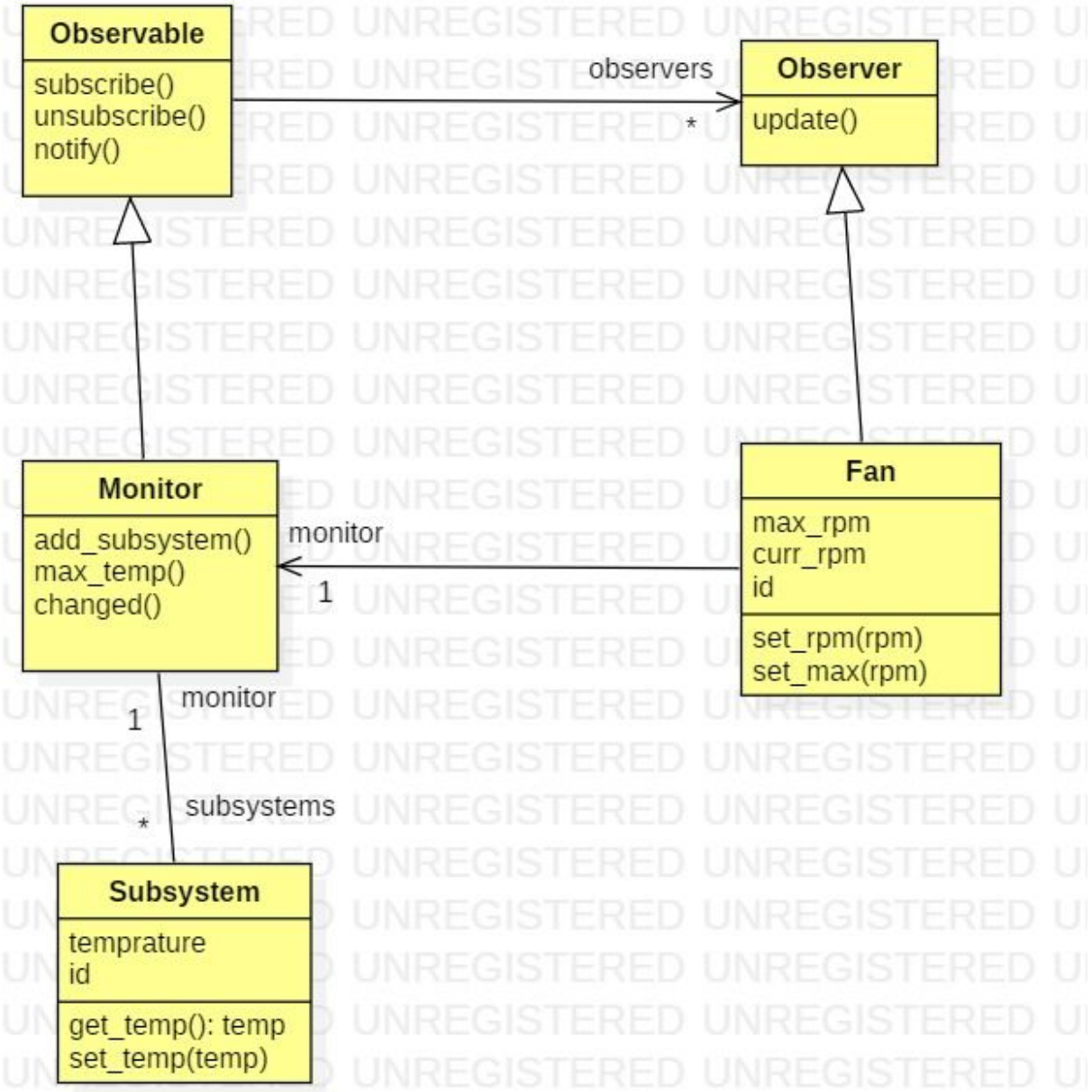
Aim: This document is intended to explain the detailed design of the monitoring system

Class Diagram:



Following are classes used:

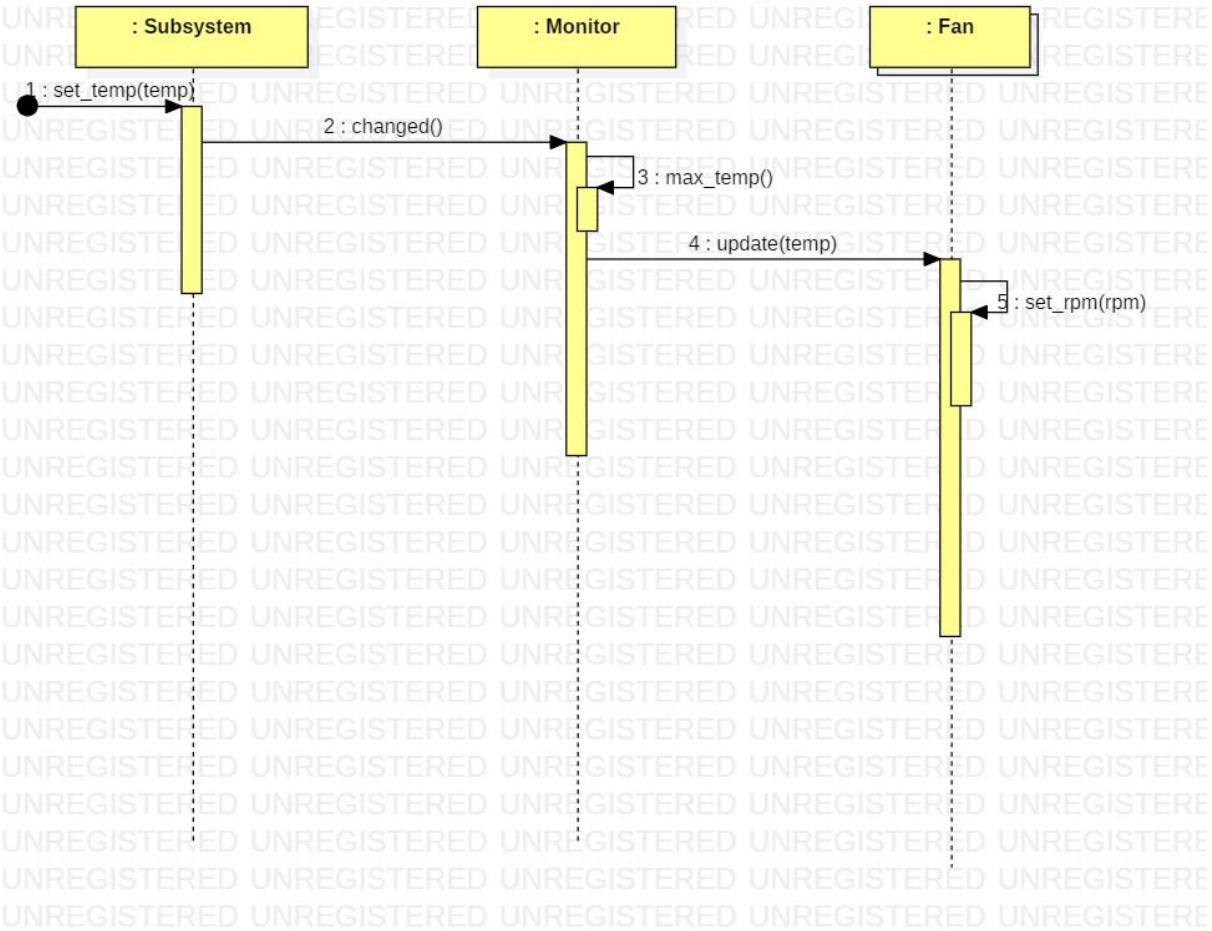
- **Fan**: This class represents a Fan entity.
    - Attributes:
        - max_rpm: It is the maximum capacity of Fan
        - curr_rpm: It is the current running speed of Fan
    - Functions:
        - set_rpm(rpm): This function sets the current running speed of the fan
        - set_max_rpm(rpm): This function changes the maximum capacity of the fan
- **Subsystem**: This class represent Subsystems
    - Attributes:
        - Temperature: It represents the current temperature of the subsystem
    - Functions:
        - set_temp(): This function changes the temperature of the subsystem
        - get_temp(): This function return current temp of subsystem
- **Monitor**: This class represents the controlling system which has a responsibility to track the temperature of subsystems and notify fans if there is any change which should impact the speed of the fan

Design Pattern Used:
Here I am using **Observable-Observer** Pattern.
Reason for choosing this pattern is it is not efficient if every fan keeps polling the temperature of all the subsystems. Instead, whenever there are significant changes in Observable entity it notifies all the observers. It does not need to know the identity of observers. Monitoring system does not need to know about fans. Fans have to subscribe to monitors and whenever there are significant changes, it will get notified. When it is notified about changes, the update method of each observer is called and logic for the respective change is executed.

Sequence Diagram

Following is a simple **sequence diagram** understanding the flow:

Use case: Change in temperature of the subsystem:



Configuration file:

Configuration.ini file has configuration as follows:

```
[BASIC]
NO_FAN=3
NO_SUBSYSTEM=4

[BOUNDS]
FAN_UPPER_BOUND=500
SUBSYSTEM_UPPER_BOUND=500
```

BASIC configuration represents no of fan and no of subsystem when the system starts.
BOUNDS represents an upper limit for the no of fan and subsystem

I have added extra functionality to add Subsystems and Fans on runtime when the system is running.

REST calls:

Following are Resources and Supported Methods:

Resource	Method	Description	Sample

fan	GET	It returns information about all the fans	curl -X GET \
			  http://127.0.0.1:5000/fan
			
			Response:
			[
			  {
			  "curr_speed": 78,
			  "index": 0,
			  "max_speed": 130
			  },
			  {
			  "curr_speed": 72,
			  "index": 1,
			  "max_speed": 120
			  },
			  {
			  "curr_speed": 78,
			  "index": 2,
			  "max_speed": 130
			  },
			  {
			  "curr_speed": 27,
			  "index": 3,
			  "max_speed": 45
			  }
			]

fan	PUT	It is used to update max speed of the fan	curl -X PUT \
			  http://127.0.0.1:5000/fan \
			  -H 'Content-Type: application/json' \
			  -d '{
			  "index": 1,
			  "max_speed":120
			}'

fan	POST	It is used to add a new fan to the system	curl -X POST \
			  http://127.0.0.1:5000/fan \
			  -H 'Content-Type: application/json' \
			  -d '{

			"max_speed":45 }'

subsystem	GET	It return all the subsystem information	curl -X GET \
			http://127.0.0.1:5000/subsystem
			Response
			[
			{
			"index": 0,
			"temperature": 12.5
			},
			{
			"index": 1,
			"temperature": 10.5
			},
			{
			"index": 2,
			"temperature": 32.5
			},
			{
			"index": 3,
			"temperature": 32.5
			}
			]

subsystem	PUT	It is used to update the temperature of the subsystem	curl -X PUT \ http://127.0.0.1:5000/subsystem \ -H 'Content-Type: application/json' \ -d '{ "index": 3, "temperature":32.5 }'
subsystem	POST	It is used to add a new subsystem to the system	curl -X POST \ http://127.0.0.1:5000/subsystem \ -H 'Content-Type: application/json' \ -d '{ "temperature":50 }'

Installation and Configuration:

Pre-requisite:
Python 3.6+
Flask

To Install Flask run following command:
pip install flask

Run the system:
Define no of the subsystem, fans and upper bounds in configuration.ini file
Run following command: python Server.py

Other improvement to solve the same problem:

1. If a number of the subsystem are big in number and there is latency in the monitor to find the maximum temperature from the subsystem following things can be done:
    a. We can treat each subsystem as a Kafka Producer. So all the subsystems temp will be written to Kafka queue. A monitor is Kafka Consumer which is consuming all this data and whenever a significant change happens it will report this to fans
    b. Introduce multiprocessing to retrieve the highest temperature

Note: Because of time constraint following things are not done:

1. REST API standard response format
2. REST API negative scenario handling (3xx,4xx)
3. Config file for logs
4. Commenting the code. Apologies for this. Most of the variable and method name are self explainatory