SE 311 SOFTWARE ARCHITECTURE

SMART CITY APPLICATION

Software Development Plan

Faruk Burak Gürel

Erşen Pamuk

Meltem Yanoğlu

17/05/2020

Outline

1. Introduction
2. Objective
3. Design Patterns
4. Functionalities
    4.1. Requirements
        4.1.1. Functional Requirements
        4.1.2. Non-Functional Requirements
5. Stakeholders
6. List of Figures
    6.1. UML Diagrams
7. Software Process
8. Project Staffing
9. Software Needs
10. Hardware Needs
11. Measurements
12. Project Schedule
13. Conclusion

1. INTRODUCTION

In today's world, with the development of technology, people's desire to get information about the events developing in their environment has increased. It is very important to be aware of the factors in the environment in order to facilitate our daily life and to be protected from some of the factors.So that's why we handled this project.

There is a city called Smart city which has various sensors installed everywhere. These sensors detect some environmental factors (e.g., temperature, pollution, congestion, noise) in the city. Temperature sensor notifies the user when temperature falls below 0, pollution sensor notifies the user when pollution AQI value is above 100, noise sensor notifies the user when noise by desibel is above 85db and congestion sensor notifies the user when car speed by km/h is below 10km/h.

According the informations we provide from the sensors, the sensors notifies the users. There is also Data Monitoring Service which is a team of specialized engineers and they send periodical queries to the sensors to check if they are malfunctioned or not. Engineers can send commands to reset a sensor and make it run again.

2. OBJECTIVES

- To make users available to see what events happened in the city.
- To notify users after certain conditions are met.
- To guide people who lives in the city.
- To let engineers check sensors to see if they are malfunctioning or not.

3. DESIGN PATTERNS

In this program, we used following design patterns ;

Observer pattern, Abstract Factory Pattern, Factory Pattern, Singleton Pattern, Template Pattern and Facade Pattern

Observer Pattern is used because each sensor and citizen are both observers. Citizens observe sensors and change that happens on their value while keeping a reference to them inside the class. Sensors observe observableValues that is beeing kept as composite references inside the subtype of City which is smartCityParts which has sensors of each type attached to it.

Abstract Factory is used in 2 places to make abstract factories for further expansion. For sensor abstract factory it has an abstract method called createSensor which creates and returns a sensor type. Each concrete factory which will be implemented with factory design pattern will return their sub type of object to this method,which can be temperature Sensor,pollution Sensor etc.
.

The other abstract factory is used in CityFactory class with the same intention and it creates concrete sub types of the abstract city according to the concrete factories type.

Factory Pattern is used in the same classes as abstract factories. There should be only one concrete factory for each sub type. And there can be only one instance of that class therefore they also implement the Singleton Pattern. Each factory is the concrete implementation of the Abstract Factory. Each implements the abstract method of the Abstract Factory class with different implementations and different returned values.

Then after that inside the CityFactory and SensorFactory classes we also used Facade Pattern to give the user an easy interface for creating new types of sensors and city parts. They don't have to know about the methods inside each class or create them by polymorphism or calling their class. They just pass in a type and name they want and the facade method checks the type and creates the wanted class without complication.

Than inside the Sensor class we implemented Template Pattern. The abstract class Sensor has a Template method which defines the skeleton of the algorithm. Each concrete class will change this methods for their type but they

won’t need to write the template method again and again.Each one of their methods will implement the primitive methods inside the abstract class and also override some of the methods inside the Sensor abstract class. Therefore it also has primitive methods which are just emtpy methods that do nothing and needs to be overriden in the subclasses.

4. FUNCTIONALITIES

The Smart City application gets information from the sensors which installed everywhere. these sensors detect some environmental factors (e.g., temperature, pollution, congestion, noise)

Besides the functional requirements, there are some nonfunctional requirements too.

4.1 REQUIREMENTS

4.1.1 Functional Requirements

REQ.	FUNCTIONAL REQUIREMENTS	PRIORITY
1	Sensors should be installed on apartment and poles in every neighborhood.	1
2	Application should get information from sensors.	2
3	User should get notified when the temperature falls below 0 degrees.	3
4	User should get notified when pollution AQI value is above 100.	3
5	User should get notified when the noise level is above 85dB.	3
6	User should get notified when the car speed is below 10km/hr.	3
7	Application should check the sensors if they are malfunctioning.	4
8	Data Monitoring Division engineers should be able to reset sensors. Either individually or all of them at once.	5

4.1.2 Non-Functional Requirements

REQ	NON-FUNCTIONAL REQUIREMENTS	PRIORITY
2	Security should be taken care of for buffer overflow and individual datas.	2
4	The program should be viable for patches.It should be maintable.	3
6	The usability of the program should be important for the end users. But as it	4

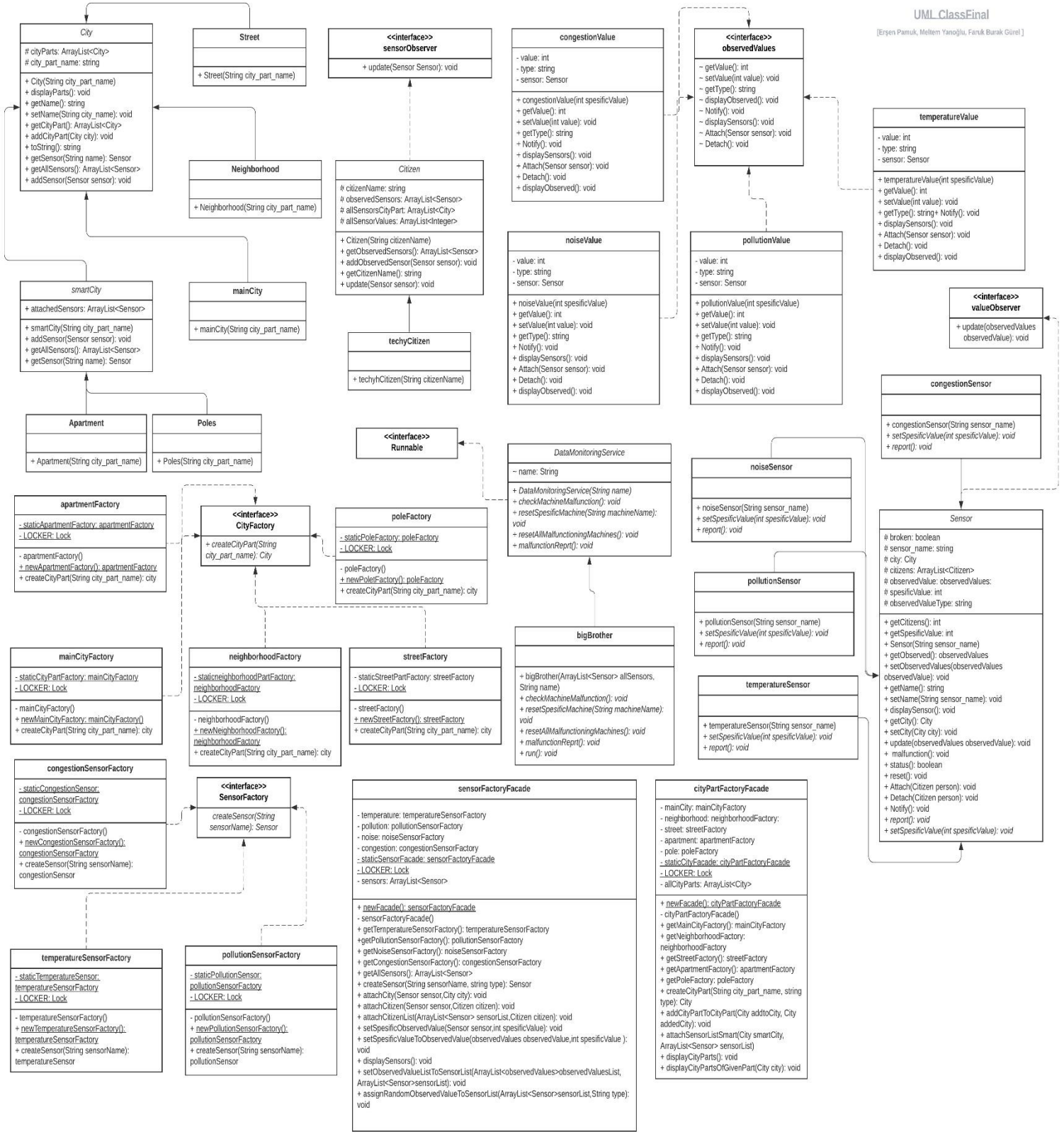
5. STAKEHOLDERS

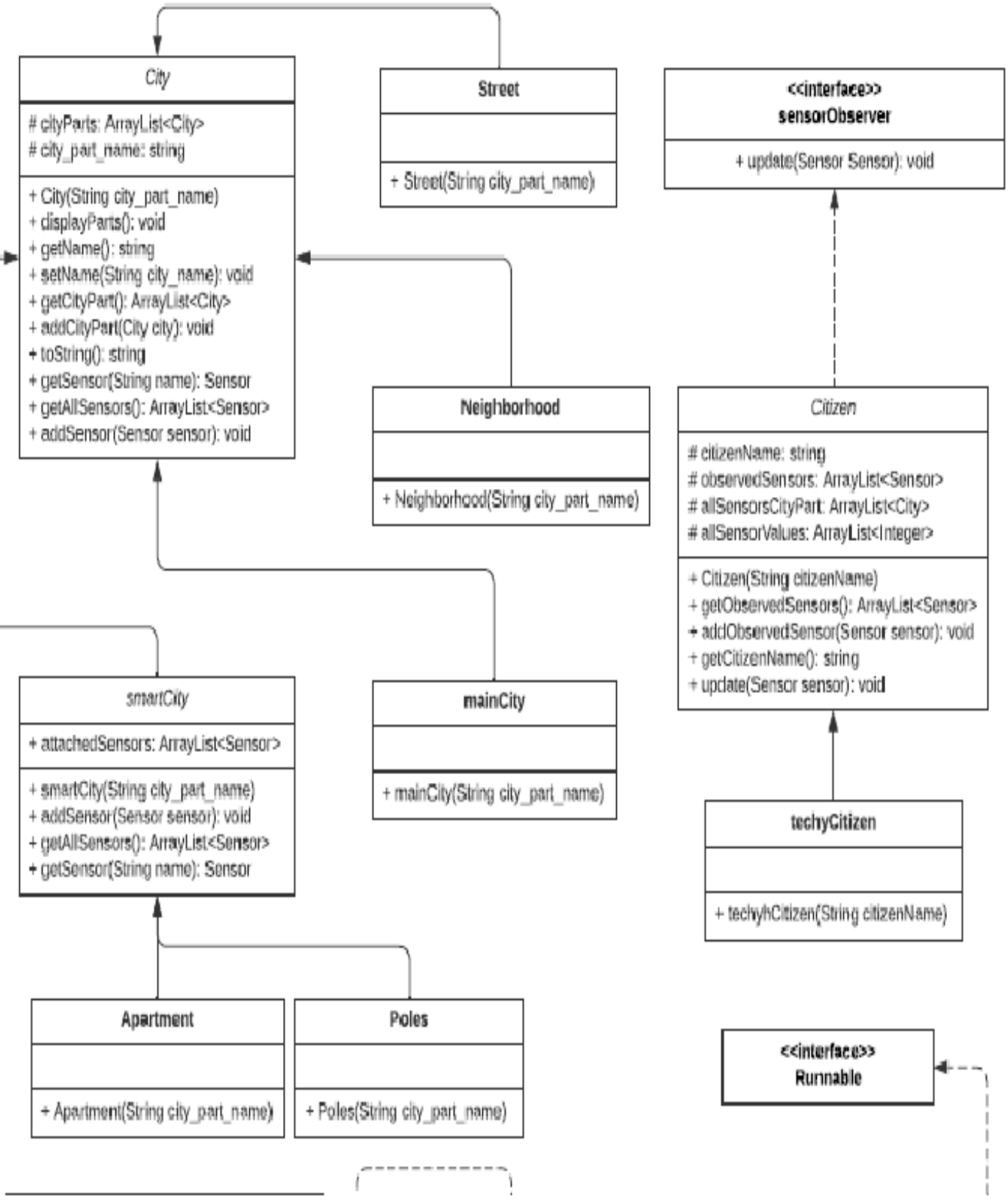
Smart city application can be used by any of the citizens that want to know about their enviroment and wants to be notified when certain event of interest happens.

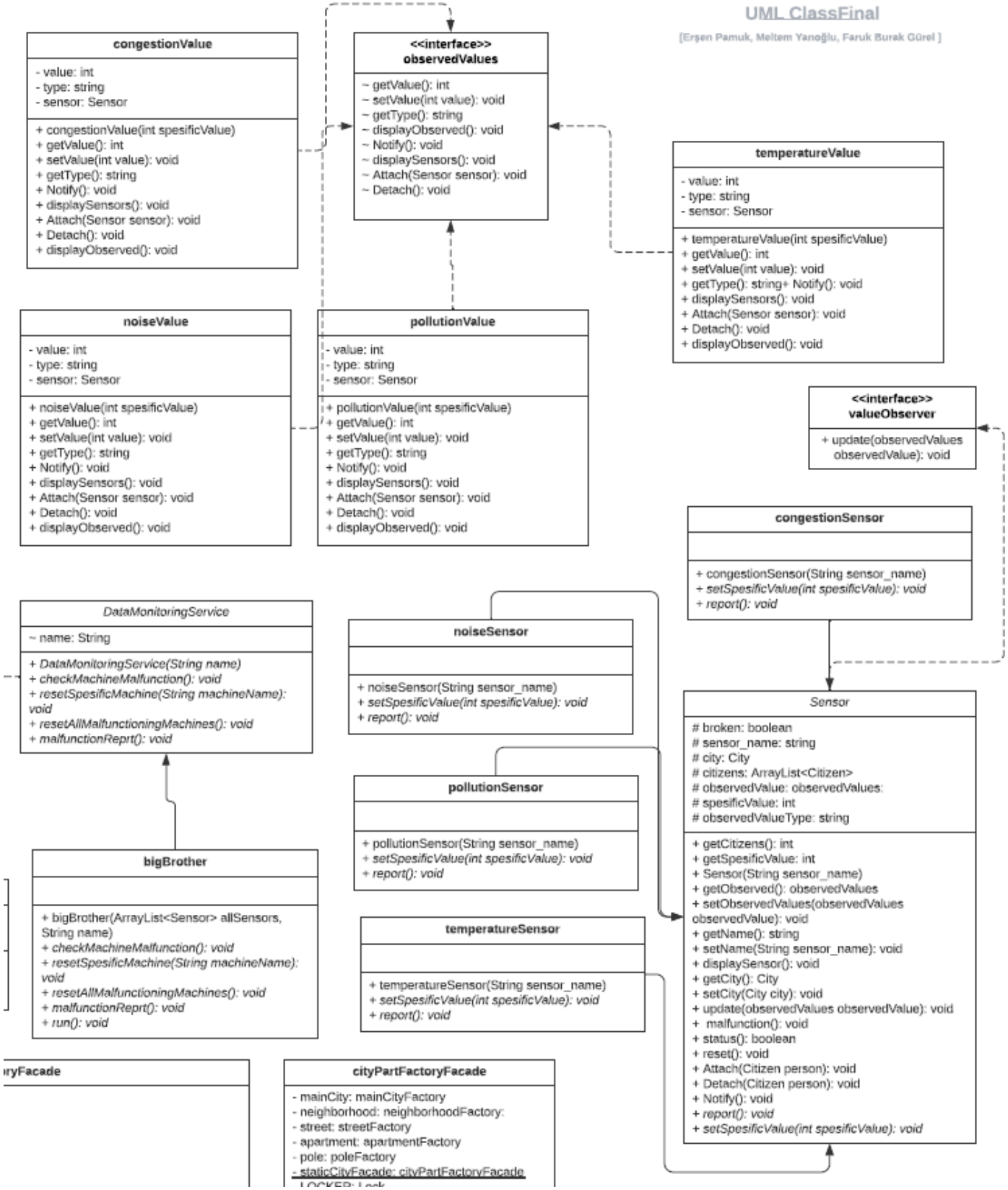
STAKEHOLDER	DESCRIPTION
CITIZEN	Any citizen living in this city can use this application to be informed about the certain event in the smart city.
TECHNICIAN	Employees who are responsible from technical maintenance, necessarily arrangements etc.

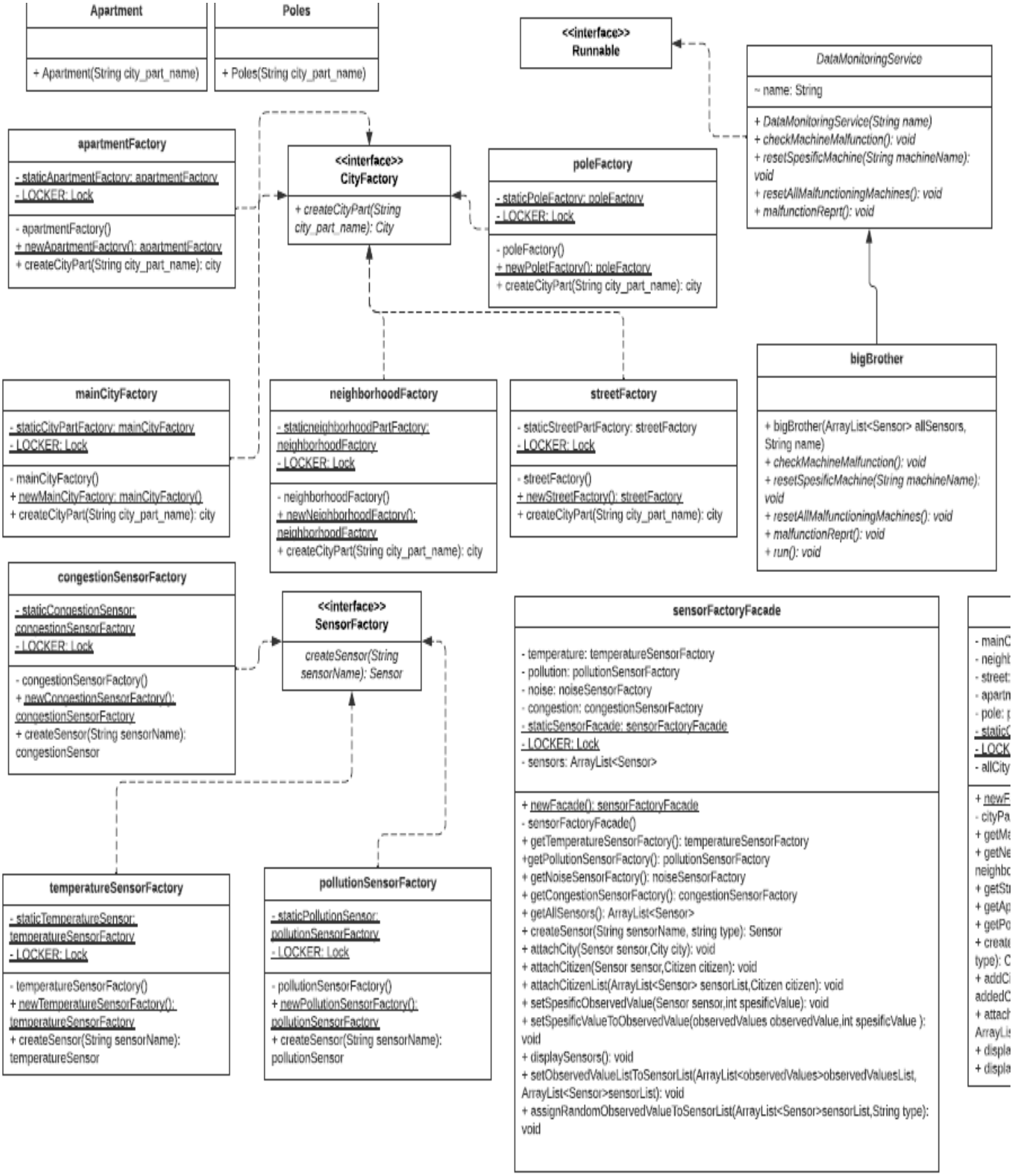
6. LIST OF FIGURES

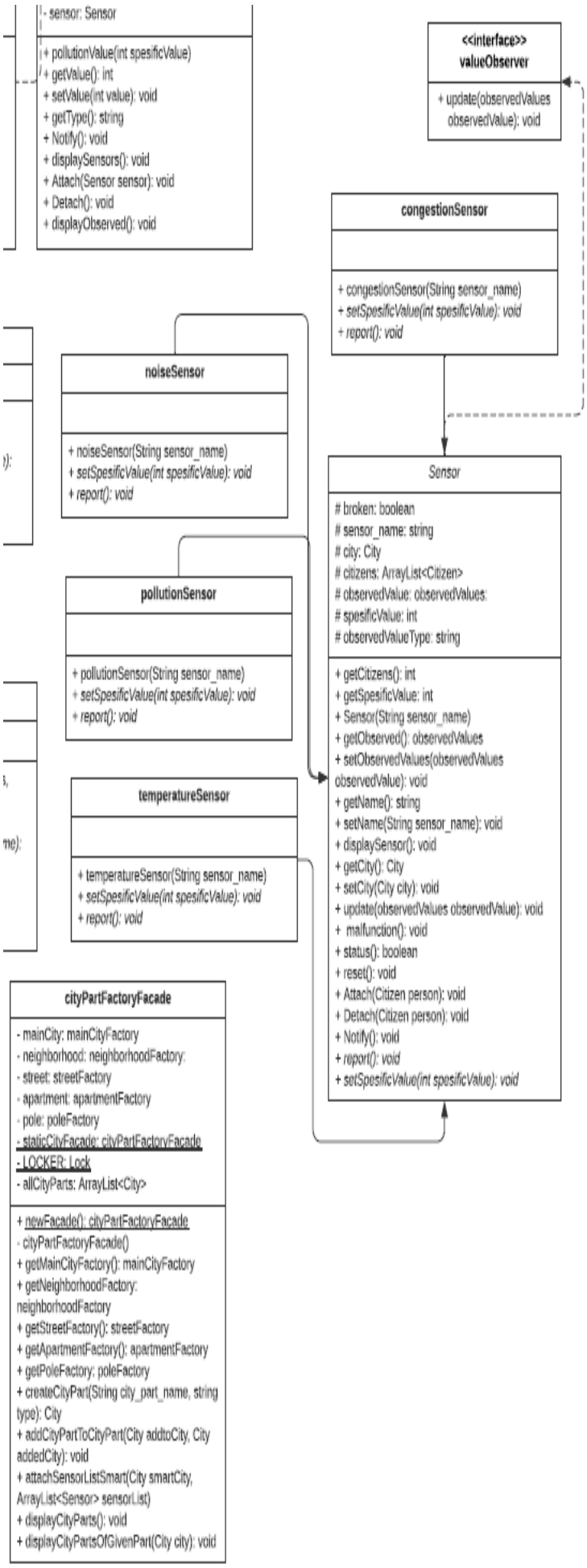
6.1.1 UML Diagrams




City
cityParts: ArrayList<City>
city_part_name: string
+ City(String city_part_name)
+ displayParts(): void
+ getName(): string
+ setName(String city_name): void
+ getCityPart(): ArrayList<City>
+ addCityPart(City city): void
+ toString(): string
+ getSensor(String name): Sensor
+ getAllSensors(): ArrayList<Sensor>
+ addSensor(Sensor sensor): void
Street
+ Street(String city_part_name)
<<interface>>
sensorObserver
+ update(Sensor Sensor): void
Neighborhood
+ Neighborhood(String city_part_name)
Citizen
citizenName: string
observedSensors: ArrayList<Sensor>
allSensorsCityPart: ArrayList<City>
allSensorValues: ArrayList<Integer>
+ Citizen(String citizenName)
+ getObservedSensors(): ArrayList<Sensor>
+ addObservedSensor(Sensor sensor): void
+ getCitizenName(): string
+ update(Sensor sensor): void
smartCity
+ attachedSensors: ArrayList<Sensor>
+ smartCity(String city_part_name)
+ addSensor(Sensor sensor): void
+ getAllSensors(): ArrayList<Sensor>
+ getSensor(String name): Sensor
mainCity
+ mainCity(String city_part_name)
techyCitizen
+ techyhCitizen(String citizenName)
Apartment
+ Apartment(String city_part_name)
Poles
+ Poles(String city_part_name)
<<interface>>
Runnable


UML ClassFinal
[Erşen Pamuk, Meltem Yanoğlu, Faruk Burak GÜrel]
congestionValue
- value: int
- type: string
- sensor: Sensor
+ congestionValue(int spesificValue)
+ getValue(): int
+ setValue(int value): void
+ getType(): string
+ Notify(): void
+ displaySensors(): void
+ Attach(Sensor sensor): void
+ Detach(): void
+ displayObserved(): void
<<interface>>
observedValues
~ getValue(): int
~ setValue(int value): void
~ getType(): string
~ displayObserved(): void
~ Notify(): void
~ displaySensors(): void
~ Attach(Sensor sensor): void
~ Detach(): void
temperatureValue
- value: int
- type: string
- sensor: Sensor
+ temperatureValue(int spesificValue)
+ getValue(): int
+ setValue(int value): void
+ getType(): string+ Notify(): void
+ displaySensors(): void
+ Attach(Sensor sensor): void
+ Detach(): void
+ displayObserved(): void
noiseValue
- value: int
- type: string
- sensor: Sensor
+ noiseValue(int spesificValue)
+ getValue(): int
+ setValue(int value): void
+ getType(): string
+ Notify(): void
+ displaySensors(): void
+ Attach(Sensor sensor): void
+ Detach(): void
+ displayObserved(): void
pollutionValue
- value: int
- type: string
- sensor: Sensor
+ pollutionValue(int spesificValue)
+ getValue(): int
+ setValue(int value): void
+ getType(): string
+ Notify(): void
+ displaySensors(): void
+ Attach(Sensor sensor): void
+ Detach(): void
+ displayObserved(): void
<<interface>>
valueObserver
+ update(observedValues observedValue): void
congestionSensor
+ congestionSensor(String sensor_name)
+ setSpesificValue(int spesificValue): void
+ report(): void
DataMonitoringService
~ name: String
+ DataMonitoringService(String name)
+ checkMachineMalfunction(): void
+ resetSpesificMachine(String machineName): void
+ resetAllMalfunctioningMachines(): void
+ malfunctionReprt(): void
noiseSensor
+ noiseSensor(String sensor_name)
+ setSpesificValue(int spesificValue): void
+ report(): void
Sensor
broken: boolean
sensor_name: string
city: City
citizens: ArrayList<Citizen>
observedValue: observedValues:
spesificValue: int
observedValueType: string
+ getCitizens(): int
+ getSpesificValue: int
+ Sensor(String sensor_name)
+ getObserved(): observedValues
+ setObservedValues(observedValues observedValue): void
+ getName(): string
+ setName(String sensor_name): void
+ displaySensor(): void
+ getCity(): City
+ setCity(City city): void
+ update(observedValues observedValue): void
+ malfunction(): void
+ status(): boolean
+ reset(): void
+ Attach(Citizen person): void
+ Detach(Citizen person): void
+ Notify(): void
+ report(): void
+ setSpesificValue(int spesificValue): void
pollutionSensor
+ pollutionSensor(String sensor_name)
+ setSpesificValue(int spesificValue): void
+ report(): void
bigBrother
+ bigBrother(ArrayList<Sensor> allSensors, String name)
+ checkMachineMalfunction(): void
+ resetSpesificMachine(String machineName): void
+ resetAllMalfunctioningMachines(): void
+ malfunctionReprt(): void
+ run(): void
temperatureSensor
+ temperatureSensor(String sensor_name)
+ setSpesificValue(int spesificValue): void
+ report(): void
cityPartFactoryFacade
- mainCity: mainCityFactory
- neighborhood: neighborhoodFactory:
- street: streetFactory
- apartment: apartmentFactory
- pole: poleFactory
- staticCityFacade: cityPartFactoryFacade


Apartment
+ Apartment(String city_part_name)
Poles
+ Poles(String city_part_name)
<<interface>>
Runnable
DataMonitoringService
~ name: String
+ DataMonitoringService(String name)
+ checkMachineMalfunction(): void
+ resetSpesificMachine(String machineName): void
+ resetAllMalfunctioningMachines(): void
+ malfunctionReprt(): void
apartmentFactory
- staticApartmentFactory: apartmentFactory
- LOCKER: Lock
- apartmentFactory()
+ newApartmentFactory(): apartmentFactory
+ createCityPart(String city_part_name): city
<<interface>>
CityFactory
+ createCityPart(String city_part_name): City
poleFactory
- staticPoleFactory: poleFactory
- LOCKER: Lock
- poleFactory()
+ newPoletFactory(): poleFactory
+ createCityPart(String city_part_name): city
bigBrother
+ bigBrother(ArrayList<Sensor> allSensors, String name)
+ checkMachineMalfunction(): void
+ resetSpesificMachine(String machineName): void
+ resetAllMalfunctioningMachines(): void
+ malfunctionReprt(): void
+ run(): void
mainCityFactory
- staticCityPartFactory: mainCityFactory
- LOCKER: Lock
- mainCityFactory()
+ newMainCityFactory: mainCityFactory()
+ createCityPart(String city_part_name): city
neighborhoodFactory
- staticneighborhoodPartFactory: neighborhoodFactory
- LOCKER: Lock
- neighborhoodFactory()
+ newNeighborhoodFactory(): neighborhoodFactory
+ createCityPart(String city_part_name): city
streetFactory
- staticStreetPartFactory: streetFactory
- LOCKER: Lock
- streetFactory()
+ newStreetFactory(): streetFactory
+ createCityPart(String city_part_name): city
congestionSensorFactory
- staticCongestionSensor: congestionSensorFactory
- LOCKER: Lock
- congestionSensorFactory()
+ newCongestionSensorFactory(): congestionSensorFactory
+ createSensor(String sensorName): congestionSensor
<<interface>>
SensorFactory
createSensor(String sensorName): Sensor
sensorFactoryFacade
- temperature: temperatureSensorFactory
- pollution: pollutionSensorFactory
- noise: noiseSensorFactory
- congestion: congestionSensorFactory
- staticSensorFacade: sensorFactoryFacade
- LOCKER: Lock
- sensors: ArrayList<Sensor>
+ newFacade(): sensorFactoryFacade
- sensorFactoryFacade()
+ getTemperatureSensorFactory(): temperatureSensorFactory
+getPollutionSensorFactory(): pollutionSensorFactory
+ getNoiseSensorFactory(): noiseSensorFactory
+ getCongestionSensorFactory(): congestionSensorFactory
+ getAllSensors(): ArrayList<Sensor>
+ createSensor(String sensorName, string type): Sensor
+ attachCity(Sensor sensor,City city): void
+ attachCitizen(Sensor sensor,Citizen citizen): void
+ attachCitizenList(ArrayList<Sensor> sensorList,Citizen citizen): void
+ setSpesificObservedValue(Sensor sensor,int spesificValue): void
+ setSpesificValueToObservedValue(observedValues observedValue,int spesificValue): void
+ displaySensors(): void
+ setObservedValueListToSensorList(ArrayList<observedValues>observedValuesList, ArrayList<Sensor>sensorList): void
+ assignRandomObservedValueToSensorList(ArrayList<Sensor>sensorList,String type): void
temperatureSensorFactory
- staticTemperatureSensor: temperatureSensorFactory
- LOCKER: Lock
- temperatureSensorFactory()
+ newTemperatureSensorFactory(): temperatureSensorFactory
+ createSensor(String sensorName): temperatureSensor
pollutionSensorFactory
- staticPollutionSensor: pollutionSensorFactory
- LOCKER: Lock
- pollutionSensorFactory()
+ newPollutionSensorFactory(): pollutionSensorFactory
+ createSensor(String sensorName): pollutionSensor


- sensor: Sensor
+ pollutionValue(int spesificValue)
+ getValue(): int
+ setValue(int value): void
+ getType(): string
+ Notify(): void
+ displaySensors(): void
+ Attach(Sensor sensor): void
+ Detach(): void
+ displayObserved(): void
<<interface>>
valueObserver
+ update(observedValues observedValue): void
congestionSensor
+ congestionSensor(String sensor_name)
+ setSpesificValue(int spesificValue): void
+ report(): void
noiseSensor
+ noiseSensor(String sensor_name)
+ setSpesificValue(int spesificValue): void
+ report(): void
pollutionSensor
+ pollutionSensor(String sensor_name)
+ setSpesificValue(int spesificValue): void
+ report(): void
temperatureSensor
+ temperatureSensor(String sensor_name)
+ setSpesificValue(int spesificValue): void
+ report(): void
Sensor
broken: boolean
sensor_name: string
city: City
citizens: ArrayList<Citizen>
observedValue: observedValues
spesificValue: int
observedValueType: string
+ getCitizens(): int
+ getSpesificValue: int
+ Sensor(String sensor_name)
+ getObserved(): observedValues
+ setObservedValues(observedValues observedValue): void
+ getName(): string
+ setName(String sensor_name): void
+ displaySensor(): void
+ getCity(): City
+ setCity(City city): void
+ update(observedValues observedValue): void
+ malfunction(): void
+ status(): boolean
+ reset(): void
+ Attach(Citizen person): void
+ Detach(Citizen person): void
+ Notify(): void
+ report(): void
+ setSpesificValue(int spesificValue): void
cityPartFactoryFacade
- mainCity: mainCityFactory
- neighborhood: neighborhoodFactory:
- street: streetFactory
- apartment: apartmentFactory
- pole: poleFactory
- staticCityFacade: cityPartFactoryFacade
- LOCKER: Lock
- allCityParts: ArrayList<City>
+ newFacade(): cityPartFactoryFacade
- cityPartFactoryFacade()
+ getMainCityFactory(): mainCityFactory
+ getNeighborhoodFactory: neighborhoodFactory
+ getStreetFactory(): streetFactory
+ getApartmentFactory(): apartmentFactory
+ getPoleFactory: poleFactory
+ createCityPart(String city_part_name, string type): City
+ addCityPartToCityPart(City addtoCity, City addedCity): void
+ attachSensorListSmart(City smartCity, ArrayList<Sensor> sensorList)
+ displayCityParts(): void
+ displayCityPartsOfGivenPart(City city): void

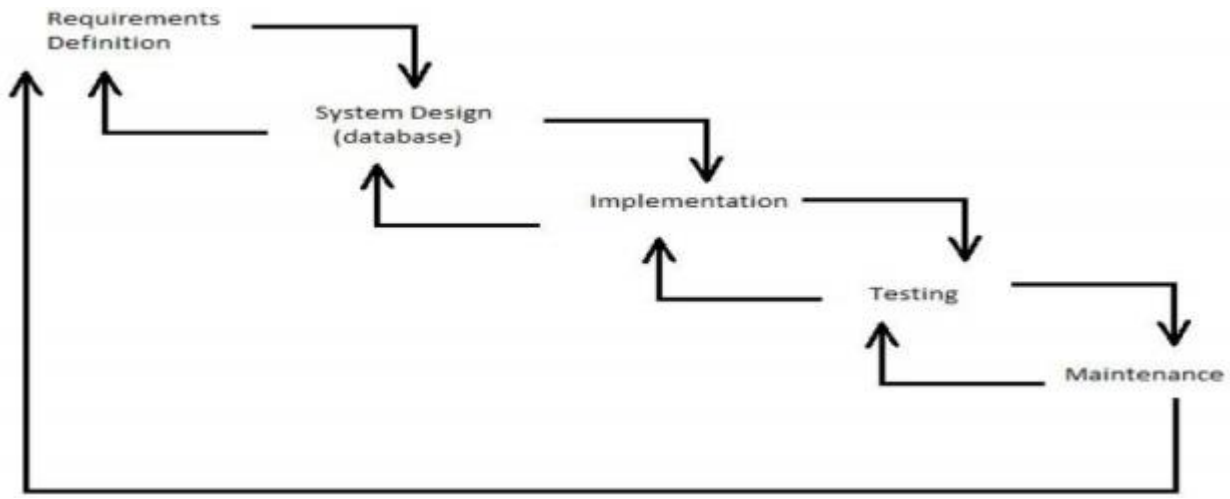
7. SOFTWARE PROCESS:

Software Process Name: The Waterfall Model

Software Process Description:

The waterfall model was the first process model to be introduced. It is also refers to as a life cycle model. It is very easy to understand and use. In a waterfall model, each phase must be completed phase by phase unless waterfall model wouldn't be durable. The waterfall model is the earliest SDLC approach that was used for software development. Because SDLC model to be used widely in software engineering to ensure success of the project.

Software Process Model:



Reasons to Choose This Model:

In the beginning, when we started to project, we should have know the requirement definition. So if we decide to requirement and defined project, we can finish to project easily and succeedly. When we completed the requirements definition, the system and design becomes durable. After we designed the system we forwarded to implementation phase. According to our system design we implemented the code and then we proceeded the testing phase we tested the program individually.

8. PROJECT STAFFING

Project Manager/Tech Lead:

Faruk Burak Gürel will be the project manager for this project. Every code before it is finalized will be inspected and refactored by the tech lead. Design patterns theoratical implementation, checking if they are correct or not,the run time memory usage shrinking and the variable naming/method naming will be done by the tech lead. The tech lead will look inside the project, make the basic classes and after that gives how the implementation will be to the coders.

Requirement Engineer:

Meltem and Erşen will be responsible about requirements, which are about identifying the stakeholder, analyzing and documenting the software requirements.

Coder and Designer:

Every person in our project is responsible about implementation and design.

9.SOFTWARE NEEDS

Eclipse: We need to use it for implementation and compiling.

Linux: We need to use debian based operating system to debug better and faster with certain software capabilities of linux.

10. HARDWARE NEEDS

Computer: A computer which has powerful hardware with qualified computer parts such as keyboards, monitors etc...

11. MEASUREMENTS

Effort and Schedule: Time spending for each group member is a measurement for our project.

Lines of Code: This reflects development progress and productivity. It also predicts future development.

Defect Count: This indicates how well the system is implemented and how effective the testing process is.

Number of Changes: It monitors changes carefully.

12.PROJECT SCHEDULE

Task Names:

a) Starting the project(1 day)

b) Defining the requirements(14 days)

c) Determining the process model(7days)

d) Tool Selection(7 days)

e) Risk Analysis(14 days)

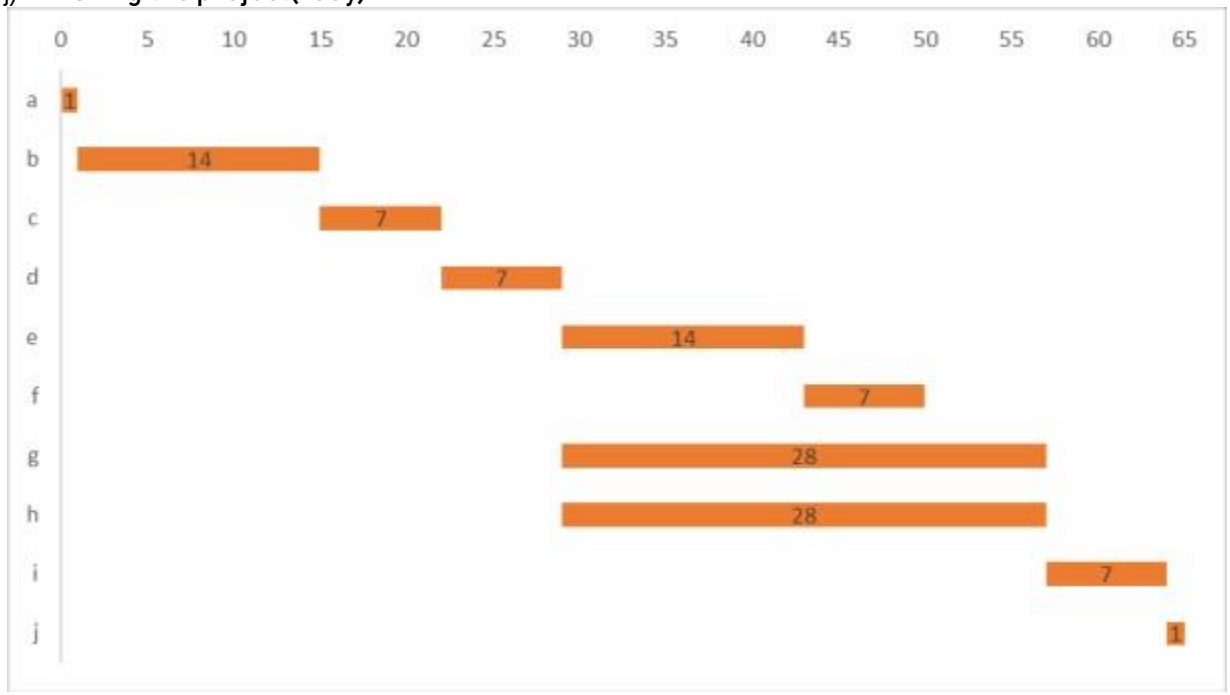
f) SPO(7 days)

g) Design(28 days)

h) Implementation(28 days)

i) Testing(7 days)

j) Finishing the project(1 day)



13. CONCLUSION

Smart City Application

The program is running smoothly with all of the needed processes,methods and design patterns being implemented professionally. It follows all of the Object Oriented Programming standards like SOLID,KISS and their new versions. It mainly uses the power of the object oriented programming,which is the main concept of Java.Tech Lead gave each member their work load accordingly to their skills and capabilities. Each team member did their part accordingly and after it all being checked by the tech lead some little errors were taken care of and everything runs smoothly. From now on any citizen living in a smart city can subscribe to sensors to get notified when certain events of interest happens.