Smart City Model Service Design Document

Date: 9/25/2020
Author: Mariam Gogia
Reviewer(s): Samer Akash, Chun Chao Tseng

Introduction

This document defines the design for the Smart City Model Service.

Table of Contents:

Overview

Smart City Model Service is one of the 4 modules of the Smart City System. This particular module is responsible for managing the domain entities of the Smart City System. Domain entities include the city, people, and six types of IoT devices. The IoT devices are street signs, streetlights, parking spaces, information kiosk, robot and vehicles: busses and cars. The Smart City Model Service provides an API for interacting with those people and it supports querying and updating the states of the entities.

Requirements

This section provides a summary of the requirements for the Smart City Model Service.

The Smart City Model Service supports querying and updating the states of the following objects:

- City

1. Smart City Model Services defines cities within the smart city. The system supports multiple cities. All cities have the following properties
    - Globally unique ID (e.g. city01)
    - Name (e.g. London, UK)
    - A blockchain account (Currently using a string as a placeholder, to be integrated with ledger in the future)
    - Location (object consisting of latitude and longitude)
    - Radius to specify the area encompassed by the city (radius is in kilometers)
    - Multiple IoT Devices of different types
    - Multiple people, residents or/and visitors
2. City can provide its details the requesting service
3. The list of cities in the system are maintained by Smart City Model API

- Person

1. A person is an inhabitant of the system and they are not directly assigned to specific cities. There are two types of people – residents and visitors. Residents are well known people to the system, whereas visitors are anonymous. Following properties are shared by both of them:
    - Globally unique ID
    - Biometric ID (e.g. voice print, face print, iris print)
    - Location (latitude and longitude)

In addition to following properties, residents have:
    - Name
    - Phone Number
    - The Role of the resident (adult, child, or public administrator)
    - Blockchain account
2. Person's details, except for their unique ID and biometrics can be updated upon request
3. The list of people inhabiting the system is maintained by Smart City Model Service API

- IoT Devices

1. The system currently supports 6 types of internet-connected devices which have following attributes in common:
    - Device ID (unique identifier consisting of the cityID and device ID, e.g. london:robot1)
    - Location (latitude and longitude of the device)
    - Enabled (on/off) (indicating whether device is on or off)
    - Current status indicating its availability
    - The latest event emitted from the device
    - The blockchain account (a string placeholder for now)

All IoT devices have the following input sensors:
    - Microphone
    - Camera
    - Thermometer
    - CO2 Meter

And one output build-in device:
    - Speaker

2. The IoT devices communicate through events. They can in take inhabitant's request as event and output the response. They also "sense" the environmental changes through events. The event is structured in the following way:
   - Type – indicating which sensor is the event sent to/coming from
   - Action – what action to be executed / what action was requested
   - Subject – this field is optional and only present if the interaction happens directly to/ from a person (resident or a visitor). Subject the unique id of the person interacting with the device.

Here are some examples of IoT events:

{microphone, "where is the nearest bus?" resident: alice}
{CO2, "400ppm"}

3. IoTs are able to understand people through NLP and reply back to them in natural language.
4. The list of all IoTs as well as the map indicating which IoT was assigned to which city is maintain by Smart City Model Service API

The types of IoT devices in Smart Cities are:

1. Street Sign
Provides the information for vehicles. It is immobile, unless deliberately moved. It is able to alter the message displayed on the sign. For example, "accident ahead," "speed limit 100 km/hr"

2. Information Kiosk
Through speech and displaying images, kiosk provides help to residents and visitors. For example, it can display a map to provide directions, it also supports purchasing tickets for concerts and events, this functionality will later be implemented as it requires connection with other modules of the smart city.

3. Street Light
Adjust the brightness in the city. It is immobile, unless deliberately moved.

4. Robot
Robots are mobile IoTs and help residents in everyday activities. They can help carry groceries, walk the dog, etc. They can also asset in emergencies.

5. Parking Space
This IoT device detects the presence of a vehicle on the parking spots and provides this information to people in the city by indicating its availability. If the parking space is taken, it charges the hourly rate to the account associated with the car (to be implemented).

6. Vehicles
These mobile IoT devices give rides to residents and visitors. They can be either a Bus or a Car. They both have a maximum rider capacity. Riding in a bus or a car is free for visitors but costs a fee for residents.

The API and Exceptions

Smart City Model Service

Smart City Model Service provides the access to external entities desiring to interact with the Smart City Model. It provides the service interface for managing the state of the cities, people, and IoT devices. Every request from the external source must go through this API

The API supports following queries and requests

- Defining the City
- Updating the City configuration
- Showing the City details, includes the details of people and IoT devices currently in the city
- Defining people in the system
- Updating the details of people
- Showing the details of people
- Creating/simulating sensor events
- Outputting sensor outputs from the devices
- Accessing IoT state and its latest event
- API keeps track of cities, people, and devices in the system.

API methods include auth_token which later be used by the Authenticating Module of the Smart City to provide security to the system.

Command API

Command Line Interface supports reading the script with commands and calls to appropriate commands in Smart City Model Service. It is a 'translator' of the scrip to the system. It works by blindly assembling the requested domain entities given their parameters and passing them to the Smart City Model Service API. Smart City Model Service API later makes decision to passed in objects. The Command API only participates in calling the commands requested in script with given objects. Detailed supported command syntax for Command API is provided under Class Dictionary section.

Smart City Model Exceptions

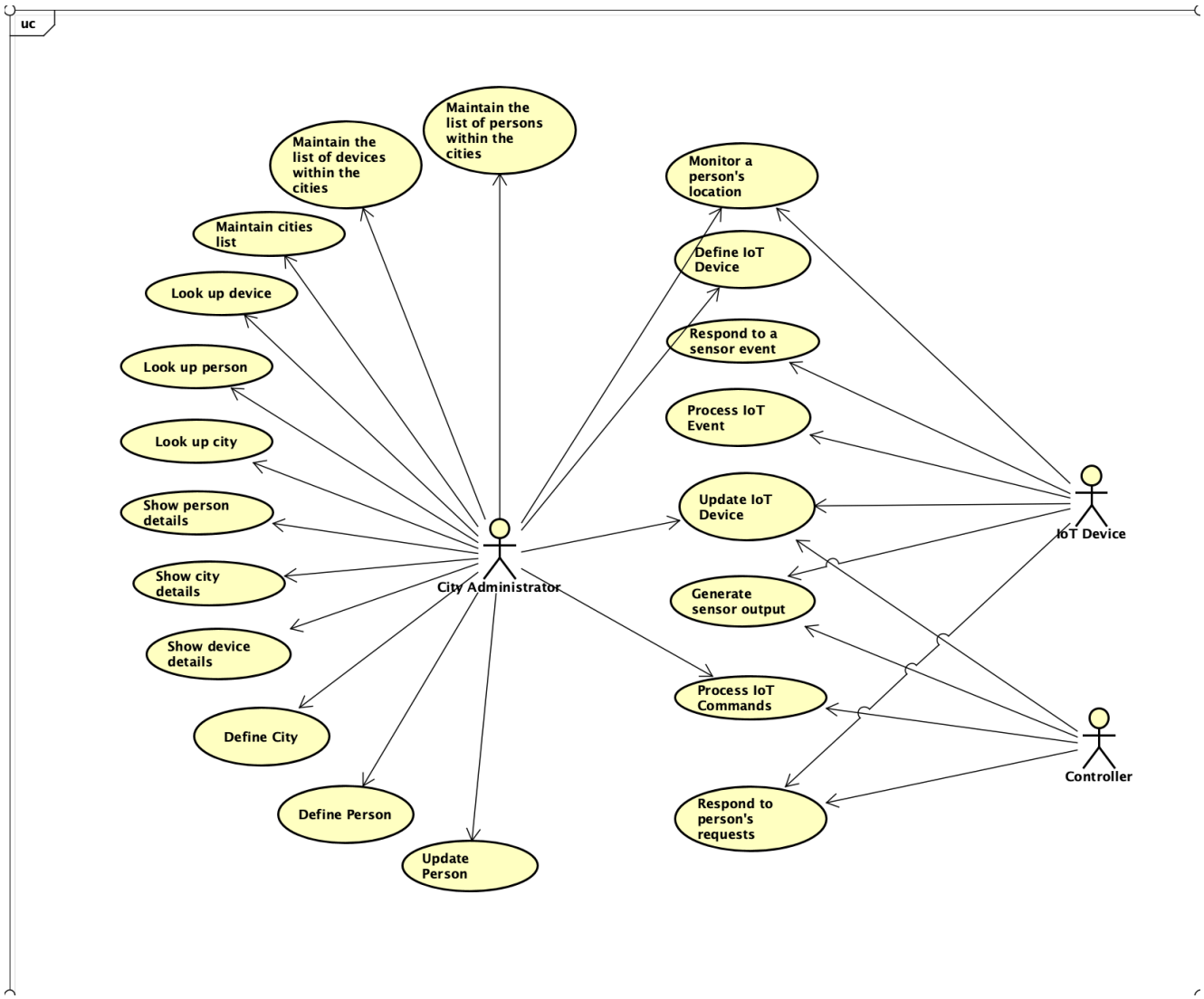
Invalid requests will result in SmartCityModelExceptions, the exception will provide the description action (e.g. method name) that caused the error and the reason the error occurred. If such exceptions occur, it will be caught and printed; it will not cause program to stop.

Command Line Interface Exceptions

Invalid command line syntax will result in CommandLineInterfaceExceptions. The exception will display the line number with the error, the command it failed to execute and the reason. CommandLineInterfaceExceptions are caught and will not cause the termination of the program.

Use Cases

The following diagram describes the use cases supported by the Smart City Model Service



Actors:

The actors of the Smart City Service Model are City Administrator, IoT Devices and a Controller Module.

City Administrator:

Responsible for querying and updating the states of IoT devices, cities, and people. Also responsible for keeping track of all the above-mentioned objects. It is the actor to be used to interact with all the domain entities and the system in general.

Controller:

Controller Module monitors sensors and controls all of the IoT devices in the system. It is responsible for monitoring the events perceived/received from sensors and responding appropriately to these events. It has several other crucial responsibilities; however, they are not relevant for the current module at the moment.

Use Cases:

Define City:
Defining city includes placing the city in the Smart City system with the unique ID, name, location, radius and the blockchain account. Ensuring that no such city exists.

Define Person:
Placing a person, either a resident or a visitor in the city with the appropriate attributes. Ensuring that no such person exists.

Define IoT:
Placing a certain type of IoT device on a specific location with attributes like device id, enabled and availability status. Depending on a type of device adding the appropriate properties, for example, if defining the street sign, provide the message that should be displayed on the screen.

Show city details:
Show the details of a city. Print out the details of the city including its id, name, account, location, details of all people currently in the city, and the details of all IoTs currently in the city.

Show device details:
Show the details of the devices. Details include enabled and availability status, location, id, and the latest event, in addition to device type specific attributes.

Show person details:
Display the details of the person, his/her location, name, and biometrics. If the person is a resident who his/her phone number, role, and the account.

Look up city:
City Administrator can find the city by its ID.

Look up person:
City Administrator can find the person by his/her ID.

Look up device:
City Administrator can find the device by its ID.

Update Person:
City Administrator can change details (e.g. phone number, location, etc.) of a person. Every detail except for the unique ID and the biometrics can be changed upon request.

Maintain cities list:
The list of cities is maintained to avoid duplications and to look up the city faster with its id.

Maintain the list of devices within the cities:
The City Administrator has access to the map of devices and cities. In key-value pair it shows which device was originally assigned to which city.

Maintain the list of persons within the cities:

The city administrator has access to the list of people, it also has an ability to determine which person is currently in which city.

Monitor a person's location:

The City Administrator, through querying can monitor person's location . IoTs can also get this information through their sensors.

Respond to a sensor event:

The IoT devices respond to sensor events. They acknowledge the event and send it to the Controller so that Controller can issue them an action command.

Process IoT event:

Once the Controller interprets the sensor event and issues the IoT event back, the IoT devices acts on the event. For example, if the event from Controller is directed to a street sign warning about the accident ahead, the street sign will display the message "accident ahead." For now, such events are simulated for the system.

Update IoT device:

All three actors are involved in this use-case. The City Administrator issues the update command, Controller evaluates the command and determines what action should IoT device take, the IoT device will then execute the action.

Generate sensor output:

This is the use case where the IoT device speaks back to people. What to say is determined by the Controller and the action is executed by the device through its speaker.

Process IoT commands:

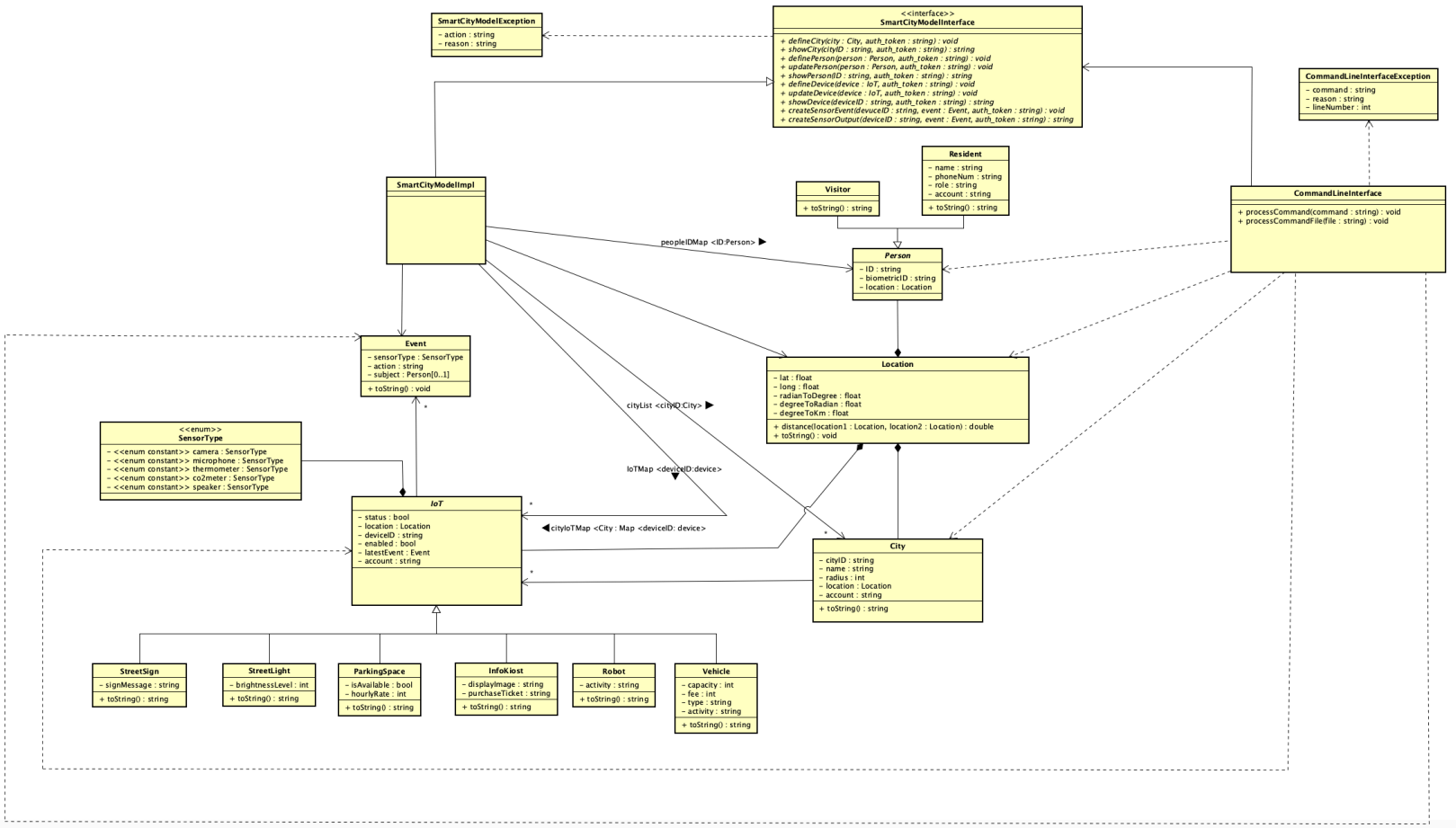
IoT commands include defining and updating devices which are processed by the City Administrator and the Controller.

Respond to person's requests:

Persons can interact with the IoTs using natural language, IoT device processes it by receiving the input and sending it over to Controller to receive the output/control command.

Implementation

Class Diagram



Interfaces and abstract classes:

- SmartCityModelInterface - defines the interface for Smart City Model Service which must act as an API for the system. It defines all the methods that the system can be accessed/queried
- Person (abstract class) - represents the abstract class for the Resident and Visitor objects, these objects share three properties which is placed in Person abstract class. The class is supposed to be extended by all types of people currently in the system and by future additions (if any) of different types of people in the city.

- IoT (abstract class) - This class hosts the attributes that all types of IoT devices must have, it also has SensorType enumerator which includes all sensors that all types of devices must have, as per requirement. If any type of device will be added to the system in the future, it should extend this class.
- CommandLineInterface - although implemented as class, it serves as an interface to the user trying to access the system via script. It reads the file, and issues commands to Smart City Model API.

Each of these following classes represent the domain entities of the system :

- City
- Resident
- Visitor
- Parking Space
- Vehicle
- Robot
- InfoKiosk
- StreetSign
- StreetLight

Helper Classes:

- Location - used as a 'helper' class to place domain entities latitude and longitude attributes into one object (location), it also hosts a method that computes the distance between two location points which is used by the Smart City Model API to determine in which cities objects (people, IoTs) are located.
- Event - Encapsulates all the attributes needed to simulate the event for IoT devices as well as to output the event coming from the IoT devices' sensors.

Exception Classes:

- CommandLineInterfaceException - if the provided script contains any command line syntax errors, e.g. not enough arguments, CommandLineInterfaceException is invoked. It provides the line number of the script that caused the error, the name of the command requested, and the reason the error occurred.
- SmartCityModelException - Exception is thrown when the given command cannot be validated/ executed, e.g. if the user asks to define the city that is already defined, if the user asks to show device that has never been defined. It displays the command name that failed and the reason it failed.

The 'API' Class:

- SmartCityModelImpl - implements all methods defined in SmartCityModelInterface, this class is used to query and update domain entities. It is the only classes through which the external source can interact with domain entities.

Dependencies, Associations, Compositions, Inheritances:

Dependencies:

1. CommandLineInterfaceException and SmartCityModelException classes depend on CommandLineInterface and SmartCityModel, respectively.

2. As CommandLineInterface class assembles domain entities before passing them to SmartCityModelImpl, therefore, it has dependencies with all domain entities and with helper classes (Location, Event).

Associations:
1. SmartCityModelService has association with all domain entities, as well with Location and the Event classes.
2. As cities have IoTs, City class is associated with the IoT class.
3. CommandLineInterface is associated with SmartCityModelInterface

Compositions:
1. Enumerator SensorType is a composite of the IoT class.
2. All domain entities, City, IoT, and Person have a composition with the Location as they cannot exist without one.

Inheritances:
1. Person is inherited by Resident and Visitor classes
2. IoT is inherited by all IoT devices: Vehicle, Robot, StreetSign, StreetLight, InfoKiosk, ParkingSpace.
3. SmartCityModelImpl inherits from SmartCityModelInterface

Class Dictionary

Smart City Model Class addresses the requirements to define, update, and show domain entities. Entities are defined in the system as per requirements - with appropriate properties and validations.

Associations

Association Name	***Type***	***Description***
peopleIDMap	Map <ID: Person>	A map of all people in the system. Using their uniqueID (String) as its key. It is used for easy look ups. It is used to update the right person in the system, used in displaying the details about the person, and validating that no such person exists when definePerson is invoked.
cityList	Map <cityID:City>	The map containing all the cities. Its unique ID is used as a key. It is used in validations and in show city command.
IoTMap	Map <deviceID:device>	The map containing all the IoTs. Its unique ID is used as a key. It is used in validations, easy look ups, in show and update device commands.

cityIoTMap	Map <City: Map <deviceID:device>	This map maps the cities with the IoT that are assigned in that city.

Methods

Method Name	***Signature***	***Description***
defineCity	(city : City, auth_token : string) : void	If the city already exists, throw SmartCityModelException. Otherwise, add it in appropriate maps.
showCity	(cityID : string, auth_token : string) : string	If the city does not exist, throw SmartCityModelException. Otherwise, return a string showing the details of the city including its current inhabitants and the IoTs.
definePerson	(person : Person, auth_token : string) : void	If the person already exists, throw SmartCityModelException. Otherwise, add the person to the appropriate map.
updatePerson	(person : Person, auth_token : string) : void	Look for the person in appropriate maps, if it does not exist throw SmartCityModelException. If found, update the appropriate fields, ensure that the existing object is updated, not the copy.
showPerson	(ID : string, auth_token : string) : string	Look for the person in an appropriate map, and return a string with details of that person. If a person does not exist, throw SmartCityModelException.
defineDevice	(device : IoT, auth_token : string) : void	Check if the device already exists, if it does, throw SmartCityModelException. Check if the device assigned city exists, if it does not, throw SmartCityModelException. Otherwise, add the IoT to the system (appropriate maps).

updateDevice	(device : IoT, auth_token : string) : void	Check if there is such device in the system, if not, throw SmartCityModelException. Determine which type of device this IoT is and update appropriately. Ensure that the already existing object is updated and not the copy.
showDevice	(deviceID : string, auth_token : string) : string	If the deviceID is given, and the device exists, return a string with the details of the device. If only the cityID is given, return the details of devices assigned to that city. Otherwise, throw SmartCityModelException.
createSensorEvent	(deviceID : string, event : Event, auth_token : string) : void	Simulates the event for IoT sensors, check if the given device exists, if the Event comes with subject, check if such subject (person) exists, set the latest event field in the appropriate device to this event. Otherwise, throw SmartCityModelException
createSensorOutput	(deviceID : string, event : Event, auth_token : string) : string	Check if the device the output is supposed to come from exists, if so, print the output, otherwise, throw SmartCityModelException

City Class - provides all attributes outlined in the requirements for the city.

Properties

Property Name	***Type***	***Description***
cityID	String	Globally unique city id
name	String	The name of the city

radius	Int	The radius of the city
location	Location	Long and lat of the city
account	String	The blockchain account of the city (currently a placeholder)

Methods

Method Name	***Signature***	***Description***
toString	():String	Provides a string with all the attributes of the city

Person
An abstract class extended by Visitor and Resident Classes, common properties for both types of inhabitants are defined in Person. These properties reflect attributes defined in requirements.

Properties

Property Name	***Type***	***Description***
ID	String	Globally unique city id
biometricID	String	Person's biometrics
Location	Location	Person's location

Visitor Class

Method Name	***Signature***	***Description***
toString	String	Provides a string with the details of a visitor

Resident Class

Properties

Property Name	***Type***	***Description***
name	String	Resident's name

role	String	A resident can be either an adult, child, or an administrator.
phoneNum	String	Resident's phone number
account	String	Resident's blockchain account (currently used as a placeholder)

Method Name	***Signature***	***Description***
toString	String	Provides the string with the details of the person

Event class

The event class is responsible for generating input and output sensor events used by SmartCityModelImpl in createSensorEvent and createSensorOutput commands .

Properties

Property Name	***Type***	***Description***
sensorType	sensorType	The type of the sensor the event is directed to is
action	String	Event action
subject	String	The optional field. The unique person ID the event is directed to/initiated by

Method Name	***Signature***	***Description***
toString	String	Provides the string containing the details of the event

Location class

Given lat, long properties of objects, location class creates the Location object. This class also has a method which determines what objects are in the city. Such functionality is needed to properly execute the requirement to 'show city' with all the objects that city currently has.

Properties

Property Name	***Type***	***Description***
lat	float	Latitude of the object
long	float	Longitude of the object
radianToDegree	float	A constant. Conversion of radians to degrees
degreeToRadian	float	A constant. Conversion of degrees to radians
degreeToKm	float	A constant. Conversion of degrees to kilometers

Methods

Method Name	***Signature***	***Description***
distance	(location1 : Location, location2 : Location) : double	Determines the distance (in kilometers) between two objects given their locations.

IoT

IoT is the abstract class for the IoT devices in the city, it is extended by 6 classes. It has a composition SensorType enumerators representing the list of sensors each device has. Sensortypes and attributes of the IoT are as per requirements for such objects in the system.

Properties

Property Name	***Type***	***Description***
deviceID	String	Globally unique id of the device
location	Location	Device's location
enabled	boolean	Shows whether or not device is on. (on/off - true/false)
status	boolean	Shows the availability status of the device (available/not available - true/false)
latestEvent	Event	The record of the event of the IoT
account	String	A blockchain account (currently a placeholder)

Enumerator
SensorType

Enumerator Component Name	***Type***	***Description***
camera	SensorType	Represents the camera sensor in IoT device
microphone	SensorType	Represents the microphone sensor in the IoT device
thermometer	SensorType	Represents the Thermometer sensor in the IoT device
co2meter	SensorType	Represents the CO2Meter sensor in the IoT device
spekaer	SensorType	Represents the built in speaker in the IoT device

Street Sign Class
Represents the street sign type of IoT.

Properties

Property Name	***Type***	***Description***
signMessage	String	Displays the message on the sign

Method Name	***Signature***	***Description***
toString	String	Provides the string containing the details of the device

Street Light Class
Represents the street light type of the IoT.

Properties

Property Name	***Type***	***Description***
brightnessLevel	Int	Adjusts the lighting in the city

Method Name	***Signature***	***Description***
toString	String	Provides the string containing the details of the device

Parking Space Class
Represents the parking space type of the IoT.

Properties

Property Name	***Type***	***Description***
isAvailable	Boolean	Determines the availability of the parking spot (available/not available: true/false)
hourlyRate	int	The hourly rate of the parking space for the residents

Method Name	***Signature***	***Description***
toString	String	Provides the string containing the details of the device

Information Kiosk Class
Represents the parking space objects of the IoT.

Properties

Property Name	***Type***	***Description***
displayImage	String	Controller should direct kiosk to display images based on persons' request
purchaseTicket	String	To be implemented later

Method Name	***Signature***	***Description***
toString	String	Provides the string containing the details of the device

Robot
Represents the robot type of the IoT.

Properties

Property Name	***Type***	***Description***

activity	String	Describes the robot's current activity

Method Name	***Signature***	***Description***
toString	String	Provides the string containing the details of the device

Vehicle
Represents the Vehicle type of the IoT.

Properties

Property Name	***Type***	***Description***
capacity	int	Outlines the maximum capacity of the vehicle
fee	int	The cost to use the vehicle
type	String	The type of the vehicle (bus or car)
activity	String	Current activity of the vehicle (e.g. "parked" ,"en route")

Method Name	***Signature***	***Description***
toString	String	Provides the string containing the details of the device

SmartCityModelException
Represents the exception class for SmartcityModelInterface

Properties

Property Name	***Type***	***Description***
action	String	Performed action (command)

reason	String	The reason it failed (e.g. object already exists)

CommandLineInterfaceException
Represents the exception class for CommandLineInterface

Properties

Property Name	***Type***	***Description***
action	String	Performed action (command)
reason	String	The reason it failed (invalid number of arguments)
lineNum	int	The line number that caused the error

CommandLineInterface

Method Name	***Signature***	***Description***
processCommand	(command : string) : void	Process a single command
processCommandFile	(file : string) : void	Process the commandFile, prepare the file to issue single commands.

Command Line Syntax
 # Define a city
 # define city <city_id> name <name> account <address> lat <float> long <float> radius <float>

 # show city <city_id>

 # Device Commands
 # Define a street sign

define street-sign <city_id>:<device_id> lat <float> long <float> enabled (true|false) text <text>

 # update a street sign
update street-sign <city_id>:<device_id> [enabled (true|false)] [text <text>]

 # Define an information kiosk

define info-kiosk <city_id>:<device_id> lat <float> long <float> enabled (true|false) image <uri>

 # Update an information kiosk
update info-kiosk <city_id>:<device_id> [enabled (true|false)] [image <uri>]

 # Define a street light
define street-light <city_id>:<device_id> lat <float> long <float> enabled (true|false) brightness <int>

 # Update a street light
update street-light <city_id>:<device_id> [enabled (true|false)] [brightness <int>]

 # Define a parking space
define parking-space <city_id>:<device_id> lat <float> long <float> enabled (true|false) rate <int>

 # Update a parking space
update parking-space <city_id>:<device_id> [enabled (true|false)] [rate <int>]

 # Define a robot
define robot <city_id>:<device_id> lat <float> long <float> enabled (true|false) activity <string>

 # Update a robot
update robot <city_id>:<device_id> [lat <float> long <float>] [enabled (true|false)] [activity <string>]

 # Define a vehicle
define vehicle <city_id>:<device_id> lat <float> long <float> enabled (true|false) type (bus|car) activity <string> capacity <int> fee <int>

 # Update a vehicle
update vehicle <city_id>:<device_id> [lat <float> long <float>] [enabled (true|false)] [activity <string>] [fee <int>]

 # Show the details of a device; if device id is omitted, show details for all devices within the city
show device <city_id>[:<device_id>]

 # Simulate a device sensor event
create sensor-event <city_id>:<device_id> type <sensorType> value <string> subject <string>

 # Send a device output
create sensor-output <city_id>[:<device_id>] type (speaker) value <string>

 # Person Commands
 # Define a new Resident
define resident <person_id> name <name> bio-metric <string> phone <phone_number> role (adult|child|administrator) lat <lat> long <long> account <account_address>

 # Update a Resident
update resident <person_id> [name <name>] [bio-metric <string>] [phone <phone_number>] [role (adult|child|administrator)] [lat <lat> long <long>] [account <account_address>]

 # Define a new Visitor

define visitor <person_id> bio-metric <string> lat <lat> long <long>

 # Update a Visitor
update visitor <person_id> [bio-metric <string>] [lat <lat> long <long>]

 # Show the details of the person
show person <person_id>

Implementation Details - How Does the system work?

The system is accessed via command script (file) that is processed by CommandLineInterface. The CommandLineInterface first reads the script, removes, comments, extra spaces, etc. and reads the script command by command.

'Define Commands'
When the command line interface identifies a define command, e.g. define city, it creates a city object with provided parameters, adds the auth_token (currently an empty string) and calls the defineCity method in SmartCityModelImpl (API). It is API's responsibility to validate the city and add it to the system. All define commands (define city, define device, define person) work in this manner. If the command is written with wrongful syntax or missing arguments, it will throw the commandLineInterfaceException. In addition to Define Commands, createSensorInput and Output commands also work this way, the Event gets created in CommandLineInterface and is passed in to the API to determine the validity and act on it.

'Update Commands'
In this implementation, client is allowed to update all the updatable fields and the order of the fields in the command line syntax is irrelevant . For example, the client can update everything about the person but its unique id and biometrics. Only the fields that need to be updated must be provided e.g., if only the phone number update is requested, the command should look like this:
update resident jane:london phone 555-555
When the update is requested, commandLineInterface is creating a new object with given parameters, if some parameters are not given, they are set to null. For example, if we are only updating janes phone number, everything except her uniqueID will be set to null. After the object creation the call to API's update method is made(e.g. updatePerosn) update function in the API will look at the uniqueID of the passed in Person object, find the person with such ID in the system, and update the found object's fields with the field values from the passed in Person object. Only the values that are not null will be copied over. This is how the system knows which field updates the client requested for the specific object. All update methods work in this manner.

'*Show Commands'*
When the show command is detected commandLineInterface will call the appropriate show function in the API, the API will send the string over which the CommandLineInterface prints.

Other Comments:
- *City overlapping is enabled because it is possible to have one city within another. For example, London has a City of Westminster.*
- *Distance and radius are both calculated in kilometers.*
- *Devices cannot be initialized using the cityID of cities that do not exist*
- *IoT devices do have the account field, however it is not utilized in this implementation and it does not*

need to be provided when initializing a certain type of IoT. This feature is to be implemented with the Ledger.
- *As per requirement parking space needs to be able to detect who's using the vehicle and charge that person's account - this requirement is not fulfilled with this implementation as it requires existence of other modules.*
- *As mentioned above, in update commands this implementation uses 'null' to determine the fields that a client wants to remain untouched. For fields with int types where 'null' cannot be used, -100 is used instead. e.g. if updating a parking space but not updating its rate, the CommandLineInterface will put -100 for the API to understand not to update this field.*

Exception Handling

- All exceptions caused by command line syntax error or illegal commands are caught by CommandLineInterfaceException
- All exceptions caused by caused by invalid request to the API, e.g. show details of the city that does not exist, is caught by SmartCityModelServiceException
- No stack trace exceptions should be expected.

Testing

The system is tested using 2 scrips: smartcity_model.script and smartcity_model2.script
smartcity_model.script a given script, however, some additional commands were added:

1. In order to test that updates were properly performed on devices, show device command was added after each device update command. Comparing before and after conditions of each type of device - tested the functionality of the update device method as well as the functionality of the define command.
2. There were several create sensor-event commands addressed to city people that do no exist, so those commands cause SmartCityModelException.
3. Added show city at the end of the file to check if the events were properly placed in appropriate 'latest-event' fields of the devices.

smartcity_model2.script
1. Tested for duplicate cities
2. Defined two cities and placed people within those cities to check if the show city function would correctly detect the list of people within correct city's radius.
3. Using 'update person' function, moved a person from one city to another to test if show city would correctly list the people within correct city's radius.
4. updated a person using different combinations of parameters to test the 'update person' function
5. update device function using different sets of parameters to update has been tested in both scrips.
6. Asked to show details of the person that does not exist to see if the exception was correctly thrown.
7. Tried to define a device with the city id that does not exist to see if the exception was correctly thrown.
8. Using update device, moved the device from one city to another to check both - the correctness of the update device function and the show city function placing the device within itself.
9. Created the event and checked if it was correctly reflected in latest-event field of the appropriate IoT.
10. Tested a parking space detecting a vehicle and changing its isAvailbale field.
11. Tried to define already existing entities to see if the exception was correctly thrown.

12. Tried to give invalid commands to see if the exceptions were correctly thrown

Both test files are commented and expected behavior is provided header to the command.

Risks

The implementation is lacking the implementation of transaction related functionalities and the security. For now, authorization tokens are set to empty strings and the entities that have account addresses are expressed as strings that act as placeholders. Also, the events and outputs are simulated, they are not based on any logic. The Controller module will provide the logic later.s