Smart City ModelDesign Documentv1.0

### Date: 04 Oct 2020

Author: Andrew Pham

Reviewer(s): Christopher Sorenson, Loi Cheng

Introduction

This document defines the design for the Smart City Model

# Overview

The current status-quo is to have each person manage their own IoT devices. However for IoT devices to work at their best, they need to be able to work together. The Smart City Model will centralize the maintenance of all IoT devices within multiple cities. The solution will enable a universal platform for IoT devices being developed for both private and public use.

The Model Service will later be integrated with the controller to send commands, a ledger to manage money, and authentication service. Together the Smart City System will manage the day to day operations of a city and support the public.

# Requirements

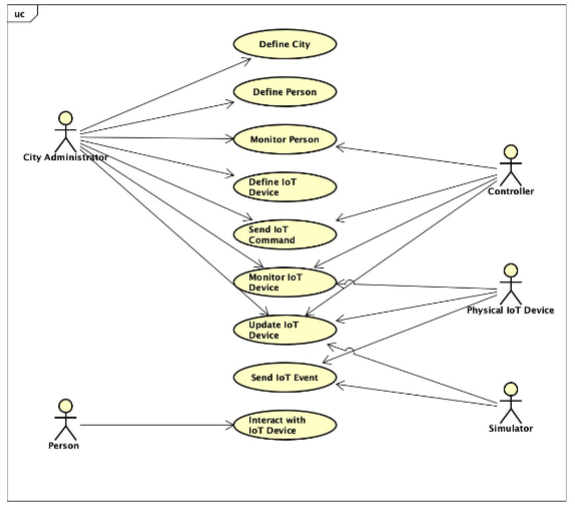
This section provides a summary of the requirements for the Smart City Model.

1. Define, update, and display the details of a city, IoT Devices, and people.

2. Controller will be able to define multiple cities.

3. Controller will be able to define devices.

4. Controller will be able to define people.

5. Controller will be able to update properties of cities.

6. Controller will be able to update properties of devices.

7. Controller will be able to update properties of people.

8. Controller will be able to send commands to devices.

9. Controller will be able to display events in a city.

10. Simulator will be able simulate events from devices.

Use Cases

This design supports the following use cases:

1. Define City

2. Define Person

3. Monitor Person

4. Define IoT Device

5. Send IoT Command

6. Monitor IoT Device

7. Update IoT Device

8. Send IoT Event

9. Interact with IoT Device

An administrator, person running the model, will be able to…

define a city, person, and devices, create them in the model.

monitor people and devices, show their states and properties

send commands, tell devices what to do

update a device, change a device’s properties.

A controller, a regulator of the model, will be able to…

monitor people and devices: show their states and properties.

send commands, tell devices what to do.

update a device, change a device’s properties.

A physical device, any non-person connected to out model, will be able to…

monitor devices, show its current states and data.

update a device, change its device properties.

send an event, post a string to the console and alert other devices in the chain.

A simulator, fake device handled by virtual device, will be able to…

update a device, change its device properties.

send an event, post a string to the console and alert other devices in the chain.

A person, anyone in the city, will be able to interact with a device, using its features and sensors.

Implementation

### The implementation has eight classes: controller, city, virtual device, device, simulator, person, event, and exception. The controller handles all of the cities, runs the commands, and prints events. Controller has one association, a list of cities. Controller has two methods, command and event. City holds the information for a city. Its properties are id, name, account, location, and radius. It has four methods getInfo, setInfo, event, and command. Cities hold people and virtual devices which are its two associations. People are either residents or visitors. They hold seven properties id, biometrics, name, phone number, role, blockchain account, and location. They have two methods getInfo and setInfo. Virtual devices have six properties id, type, location, status, enabled, and event. Virtual Devices have two methods, command and event. Each virtual device has two associations, one simulator and one physical device. Physical devices have four properties corresponding to their sensors a microphone, camera, thermometer, and co2 meter. Devices have two methods, getSensorData and command. Simulators only have one method, simulateEvent. Events have three properties type, action, and subject.

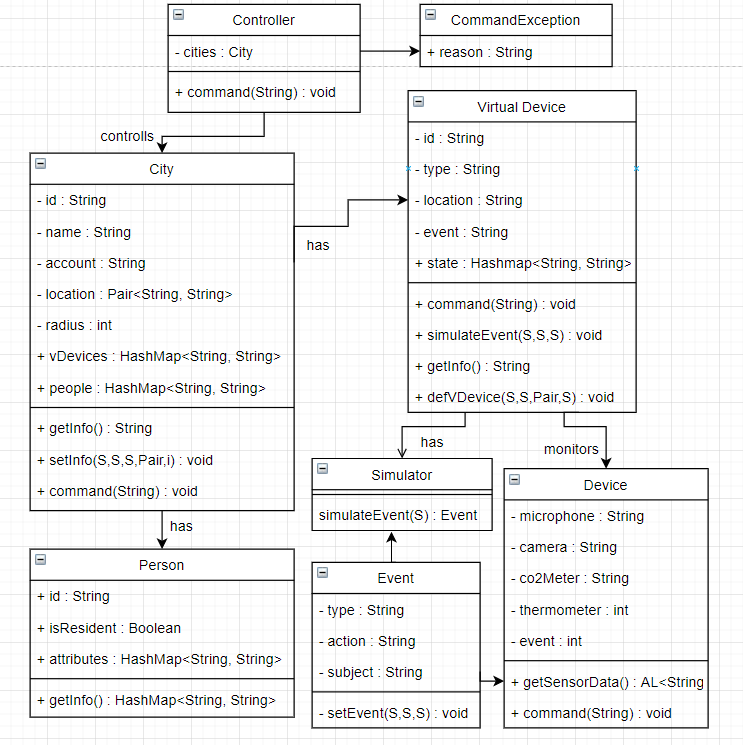
The model module fits into the overall architecture by creating the simulation of the city. It is controlled by the controller service and supported by an authentication service.

### Overall UML Component Diagram

### 

Class Diagram

The following class diagram defines the classes defined in this design.



# Class Dictionary

## Controller

The controller sends all commands to create and update cities, people, and devices. It also posts events to the console. The controller is the API for interacting with the model. We use the following syntax:

**City Commands**

# Define a city

define city <city\_id> name <name> account <address> lat <float> long <float> radius <float>

# Show the details of a city. Print out the details of the city including the id, name, account, location, people, and IoT devices.

show city <city\_id>

**Device Commands**

# Define a device

define <device\_type> <city\_id> <device\_id> [<attribute> <value>]\*

# Update a device

update <device\_type> <city\_id> <device\_id> [<attribute> <value>]+

# Show a device

show device <city\_id> <device\_id>

# Simulate a device sensor event

create sensor-event <city\_id> <device\_id> type <type> value <value> [subject <subject>]

# Send a device output

create sensor-output <city\_id> <device\_id> type <type> value <value>

**Person Commands**

# Define a new person

define <type> <city\_id> <person\_id> [<attribute> <value>]\*

# Update a person

update <type> <person\_id> <city\_id> [<attribute> <value>]+

# Show the details of the person

show person <person\_id>

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| command | Param: String  Return: void | Takes command from the console and sends it to virtual device |

***Associations***

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| cities | Map<String, City> | List of cities managed my Smart City Model. A controller has a composition of cities. |

## VirtualDevice

A virtual device records the state of the actual device, relays commands to the device, and takes in events.

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| command | Param: String  Return: void | Takes in command and sends it to device. |
| getInfo | Param: none  Return: String | Returns string with device’s info including event. |
| defVDevice | Param: String type, String id, Pair location, String enabled  Return: void | Sets virtual device’s properties |
| simulateEvent | Param: String type, String value, String subject  Return: void | Calls upon the simulator’s simulateEvent method and sets e |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| id | String | unique identifier |
| type | String | IoT device type such as sign, light, or kiosk |
| location | Pair <String, String> | Latitude and longitude |
| e | Event | Latest event from the device or simulator |
| state | Hashmap<String, String> | Keeps track of the state of the virtual device such as street sign brightness, text, etc. Also contains the requirements of status and enabeled. Example:  { [”brightness”, “10”], [”enabled”, “true] } |

***Associations***

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| d | Device | Actual physical device object. A virtual device has one physical device. |
| s | Simulator | Simulator for this device. A virtual device has one physical device. |

## Device

A physical device, has sensors and gives their readings.

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| getSensorData | Param: None  Return: ArrayList<String> | Returns a list with microphone, camera, thermometer, and CO2 data respectively. |
| command | Param: String  Return: void | Takes in and processes commands. Device outputs and actions are handled here. |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| microphone | String | Records words input to microphone |
| camera | String | Records description of picture taken |
| thermometer | int | Records temperature in F |
| co2Meter | int | Records surrounding CO2 in PPM |
| e | Event | Latest event. Can be set by sensors when that functionality comes. |

## Simulator

A simulator for a device that creates fake events.

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| simulateEvent | Param: String  Return: Event | Takes in event request and sends that simulated event to virtual device. |

## City

A city object, holds devices, people, and information about the city.

***Method***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| getInfo | Param: None  Return: String | Returns string with all properties, device and people names. |
| setInfo | Param: String id, String name, String account, Pair location, int radius  Return: void | Sets properties of city |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| id | String | Unique identifier |
| name | String | City name |
| account | String | Blockchain account address |
| location | Pair<String, String> | Latitude and longitude of center of city |
| radius | int | Radius from location to boarder of city in miles |

***Associations***

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| vDevices | Hashmap<String, String> | List of virtual devices in this city |
| people | Hashmap<String, String> | List of residents in city |

## Person

A person in the city, either a resident or visitor. Typically visitors will not have as many attributes as a resident; however, there is no need to limit them.

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| getInfo | Param: None  Return: HashMap<String, String> | Returns attributes |

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| id | String | Unique ID |
| isResident | bool | Describes if visitor or resident |
| attributes | Map<String, String> | Describes the rest of person attributes including name, number, role, account, and location. |

## Event

An event describes the type, action, and subject.

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| type | String | Type of event |
| action | String | Action recorded by device |
| subject | String | Optional subject |

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| setEvent | Param: String type, String action, String subject  Return: void | Sets the event’s fields |

## CommandException

A command exception tells us when a command is invalid.

***Properties***

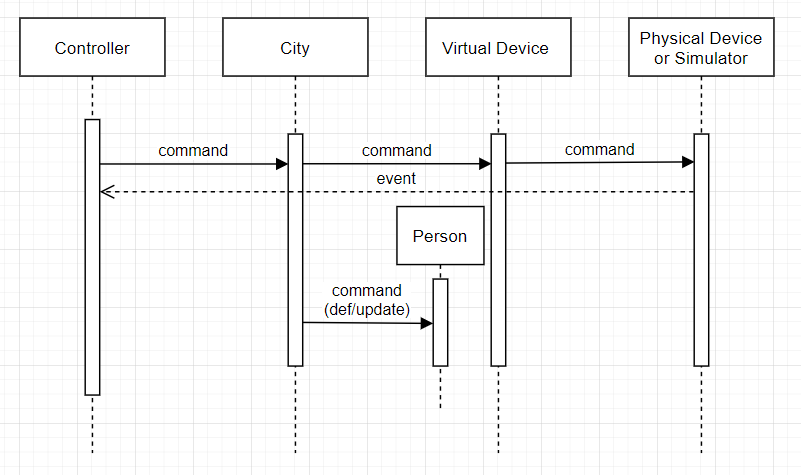
|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| reason | String | Why command was rejected |

Implementation Details

In short, we are required to be able to define, update, and show cities, people, and devices. Next we need to be able to simulate events and receive commands. A controller has cities, a city has people and devices. From the controller we can view/update cities/people/devices, send commands, and simulate events.

The parts of this implementation are fitted together in a sort of chain. A controller has cities which have people and virtual devices which has a physical device and simulator. The controller passes down a command through the same chain to the device. Definitions, updates, and commands will be handled with commands. We will monitor the details of the city (people and devices) via events through the console. Events will be simulated with the simulator. The type of event can be dictated by a command string. Events are accessed from the controller by asking the virtual device for an update on its state. This design addresses the requirements by allowing us to encapsulate parts, update properties, receive events, and send commands. By encapsulating the cities, people, and devices, we can ensure we are sending commands to specific devices in specific cities.

### UML Sequence Diagram



Exception Handling

Exceptions will be handled with an error printed to the console with no change to the state of the model. We foresee one primary type of exception: incorrect commands referencing objects that do not exist such as cities, devices, or people. The controller will ensure only correct inputs are allowed. There will be one type of exception, a command exception, since all user input comes via commands. A command exception is returned in response to an error. Exceptions have one property, reason. Reason is a string that describes why the command could not be performed. Exceptions could more specifically handle things such as limiting states and attributes to only currently known and specified definitions; however, this would hurt scalability and extendibility later if we wanted to add new devices, types, or attributes.

# Testing

Program will run a script provided by the class and another script with all possible commands. Program functions (create, update, print event) will be timed with a number of devices with each test increasing the number of devices by an order of magnitude. All tests will be re-run after any software change. Scripts will be created to test incorrect inputs. We expect an exception error for every command of this test.

# Risks

This implementation has no use of storage. A system failure would lead to a total loss of the system. This can be corrected by saving a record of the system for every update and loading it if a restart is necessary. The controller has full control over all devices and personal information in a city. Special care needs to be taken to prevent misuse, malicious or otherwise.