Smart City Controller Design Document v1.0

Date: 26 Oct 2020
Author: Andrew Pham
Reviewer(s): Pawel Nawrocki and Licheng Xu

Introduction
This document defines the design for the Smart City Controller

Overview
The Smart City Controller Service is responsible for monitoring the devices and people within the city. Also, the Controller Service can generate actions to control the devices based on rules, in response to status updates from the devices. Devices include sensors that are able to collect and share data. Devices can also be controlled. Please refer to the Smart City Model Service Requirements document for more information.[1]

Document Organization
1. Requirements
2. Use Cases
3. Implementation
4. Class Diagram
5. Class Dictionary
6. Implementation Details
7. Sequence Diagram
8. Exception handling
9. Testing
10. Risks

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

1. Monitor devices for status updates.
2. Apply rules that respond to the status updates from the device sensors and generate actions.
3. Process sensor inputs such as voice commands received via microphone.
4. Respond to events by, generating and sending command messages to Devices.
5. Process payment transactions.
6. Must be able to respond to the following stimulus with appropriate action:

    - Natural disaster: announce emergency, robots address emergency, robots help people shelter
    - Traffic accident: robot announces stay calm, robot helps
    - Co2 too high: disables cars
    - Co2 lowers again: enables cars
    - Person litters: says don’t litter, cleans, charges fine
    - Broken glass: cleans

[1] From Controller Service Requirements Document

- Sees person: update location
- Find person: tells person to stay put, gets the person from last location
- Parking: charges vehicle owner
- “Does this bus go to town square?”: says yes
- Person boards bus: says good to see you, charges person
- What movies are showing?: says Casablanca
- Reserve movie: charges for movie, says seats reserved

Use Cases

This design supports the following use cases:

A controller, the regulator of the model, will be able to create sensor events that may trigger responses.

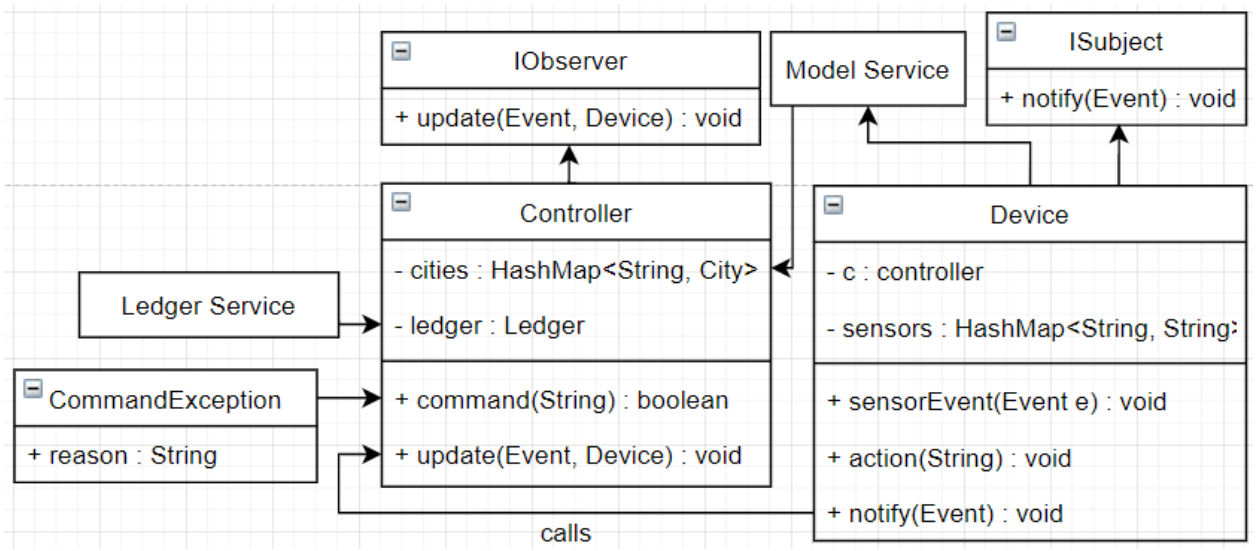
Use case diagram from previous design document.

Implementation

We implement the observer pattern design to have our subjects update our observers. In practice this means the devices with sensors update our controller which then can respond to rules with commands. In reference to the observer pattern, the observable subjects are our devices and the observer is the controller.

Class Diagram

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



A simplified class diagram omitting details about other services and some class attributes.

Class Dictionary

Controller

The controller processes commands and sends all commands to the devices. Devices that have an event will call the update method on the controller. Controller commands related to interacting with the model service is included in the appendix. All design requirements are met using the controller class; it monitors events and responds to them appropriately.

```
Syntax:

create sensor-event [city] [device] type [sensor] value [value] subject [subject]

Subject is optional.
```

Methods

Method Name	Signature	Description
command	Param: String Return: boolean	Takes command from the console and sends it to virtual device. Commands are a string,

		the command and receiver are encapsulated in the string.
update	Param: Event e Return: void	Analyzes the property to check if it sets off any rules. If a rule applies, calls command() with the proper actions.

Associations

Association Name	**Type**	**Description**
cities	Map<String, City>	List of cities managed my Smart City Model. A controller has a composition of cities.
ledger	Ledger	A ledger to keep track of transactions.

Device

A physical device has sensors and gives their readings. The device is the subject of the observer pattern. It is a part of the model service but included here for its relevance. Whenever a sensor event occurs, it calls the method notify.

Methods

Method Name	**Signature**	**Description**
sensorEvent	Param: Event e Return: void	Changes properties then calls notify method.
action	Param: String Return: void	Takes in and processes commands. Device outputs and actions are handled here.
notify	Param: Event e Return: void	Calls upon observe method in controller and passes new sensor data.

Properties

Property Name	**Type**	**Description**
id	String	Device ID
sensors	Map<String, String>	Keys are the sensors, microphone, camera, thermometer, and co2. Values are the current value of the sensors
location	Pair<String, String>	Key is long, Value is lat
city	City	City device is in
self	Device	Self-reference

Associations

Association Name	Type	Description
c	Controller	Pointer to controller so that we can call its update method in notify()

Event
An event describes the type, action, and subject.

Properties

Property Name	Type	Description
type	String	Type of event
value	String	Action recorded by device
subject	String	Optional subject

Methods

Method Name	Signature	Description
setEvent	Param: String type, String value, 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

Interfaces

Observer
A controller is an observer and implements observer. It must have an update method to process notifications from the subject it subscribes to.

Methods

Method Name	Signature	Description
update	Param: Event e, Device origin Return: void	

Subject

A device is a subject and implements subject. It must have a notify method to send notifications to its observers.

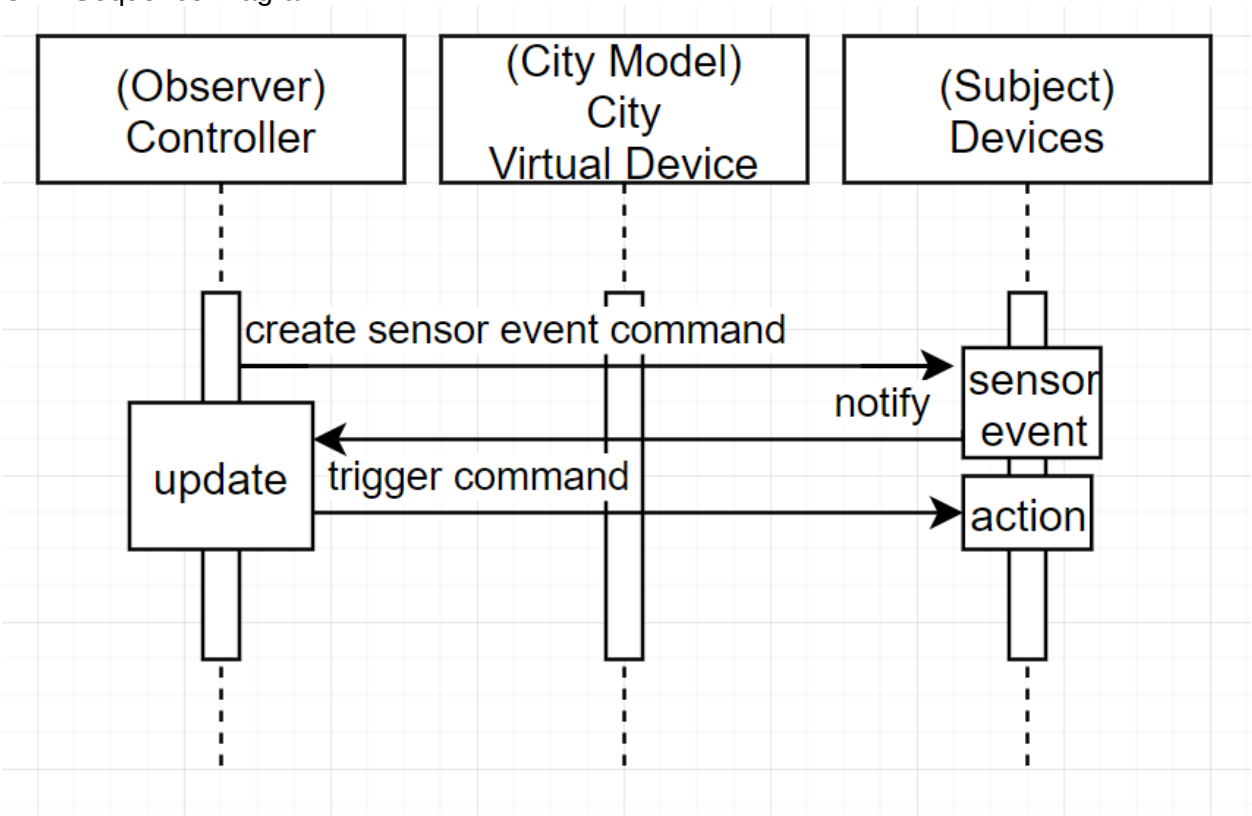
Methods

Method Name	Signature	Description
notify	Param: Event e Return: void	Calls upon observe method in controller and passes new sensor data.

Implementation Details

Usually in observer pattern we can have more than one observer subscribed to a subject; however, in our case we only need one subscriber, the controller. Only the controller needs to know what events are occurring so that it can respond appropriately. Devices will have the controller as a built in subscriber. When a device updates its state (in our case via an event created by the controller), it will notify the controller by calling upon its update command. The update command is passed the event, decides if any actions need to be taken, and if so calls the appropriate commands via the command method in the controller.

UML Sequence Diagram



A sequence diagram showing the flow of creating a sensor event by text file input, the sensor event notifying the controller of an event, and the controller triggering commands.

Exception Handling

Improper commands will throw a command exception. Ledger exceptions for insufficient funds are not thrown; instead the subject will be alerted to their failed transaction. Exceptions will be handled with an error printed to the console with no change to the state of the model. 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.

Testing

Testing will be done with the provided script and modular tests. Each modular test will test one specific component of the controller to prove its functionality. Test inputs and outputs will be saved in .txt format in the test results folder. Tests can be run by putting a test input file in the same directory and using the file path as an argument.

Test numbers and descriptions:

1. The given script provided by the class with minor functionality and syntax modifications.
2. Natural Disaster (fire)
3. Traffic Accident
4. CO2 trigger cars disable and enable
5. Littering
6. Broken Glass
7. Find Person
8. Parking
9. Bus
10. Movies

Risks

The greatest risk we foresee is the in responding to events is the problem of tasking. Currently robots are given a task without consideration for their current task. For example, if a robot is walking a dog, should it stop immediately to address a fire? In our current implementation, the device will always perform the latest task given even if they were not finished with their last task. In the future robots should create a “task-complete” event to open them up for tasking. Also urgency levels for tasks must be created from normal priority to life threatening priority. These urgency levels should allow emergency tasks to overwrite current tasks.

Appendix: model commands known by the controller

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>
```