`CSCI E-97 Software Design 10/12/2020

Smart City Controller Service, Design Document

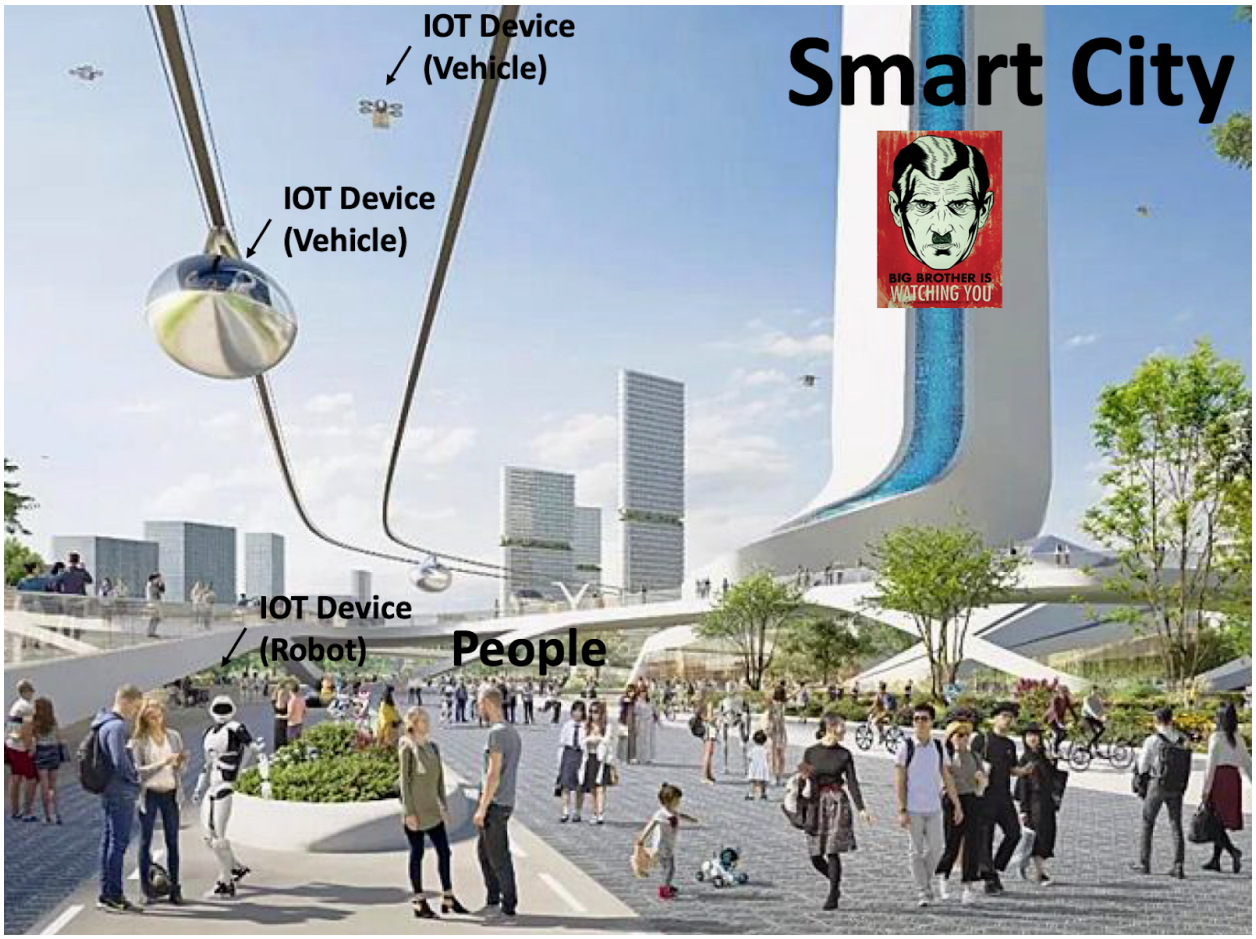
Date: 10/12/2020

Author: Loi Cheng

Reviewer(s): Eric Gieseke

Introduction

The city of the future can be automated by the Smart City Software System. through AI-powered Internet of Things (IoT) devices, smart City allows city administrators to fully automate a city. Devices include cameras, microphones, robots, and other devices. Sensors monitor the condition of people and devices. Parking spots automatically charge fees to the user's blockchain account. Robots help residents clean the city, respond to emergencies, and fight terrorists.



Caption: People and Devices in the Smart City

Overview

The implementation of a Smart City Controller is described. The controller responds to specific conditions of the city as indicated by IoT devices and sensors. Some examples of conditions include:

- Fire detected at specified location -> dispatch IoT devices to put out the fire
- CO2 levels abnormal -> disable all cars in the city
- Resident boards a bus -> charge fare to the resident account
- Person asks about movie info -> respond with info about movie showings
- Virus outbreak -> dispatch masks to all residents
- Robot uprising rebellion -> dispatch all robots still under control to quell the uprising

Requirements

This section defines the requirements for the Smart City Controller Service.
The service should support the following functions:

- Monitor Devices for status updates.
- Apply rules that respond to the status updates from the device sensors and generate actions.
- Sensor input includes voice commands received via the microphones.
- In response to events, generate and send control messages to Devices.
- All payment transactions are performed using the Blockchain Ledger with the Unit currency.

The Smart City Controller Service should use the Smart City Model Service interface to monitor the status of each of the IoT devices installed within the houses. In response to inputs, the Controller Service will use rules to invoke actions on the IoT devices. Log all rule execution and resulting actions.

Design Input:

- Follow the modularity specified in the Smart City System Architecture document.
- Apply the OBSERVER Pattern to allow the Smart City Controller to “listen” for events emitted by the Model Service Sensors.
- Apply the Command Pattern to implement the Actions performed by the Smart City Controller Service.
- Use the Ledger Service (from assignment 1) to check account balances and submit transactions for checkout.

Sensor, Stimulus, Rule, Action

The following table defines the behavior for the Controller Service. The Controller Service will monitor all device sensors for each of the cities. For each stimulus, apply the appropriate action.

Name	Sensor or Appliance	Stimulus (within the context of a city <city_id>)	Action
Emergency 1	Camera	emergency <emergency_type> at lat <lat> long <long> Where <emergency_type> is one of: fire flood earthquake severe weather	action for <city> 1. announce: "There is a <emergency_type> in <city>, please find shelter immediately" 2. ½ Robots: "address <emergency_type> at lat <lat> long <long>" 3. remaining robots: "Help people find shelter"
Emergency 2	Camera	emergency <emergency_type> at lat <lat> long <long> Where <emergency_type> is one of: traffic_accident	action for <reporting_device> 1. announce: "Stay calm, help is on its way" 2. Nearest 2 Robots: "address <emergency_type> at lat <lat> long <long>"
CO2 Event	CO2 Detector	CO2 level over 1000	If reported by more than 3 devices within a city, Disable all cars in the city.
CO2 Event	CO2 Detector	CO2 level under 1000	If reported by more than 3 devices within a city, Enable all cars in the city.

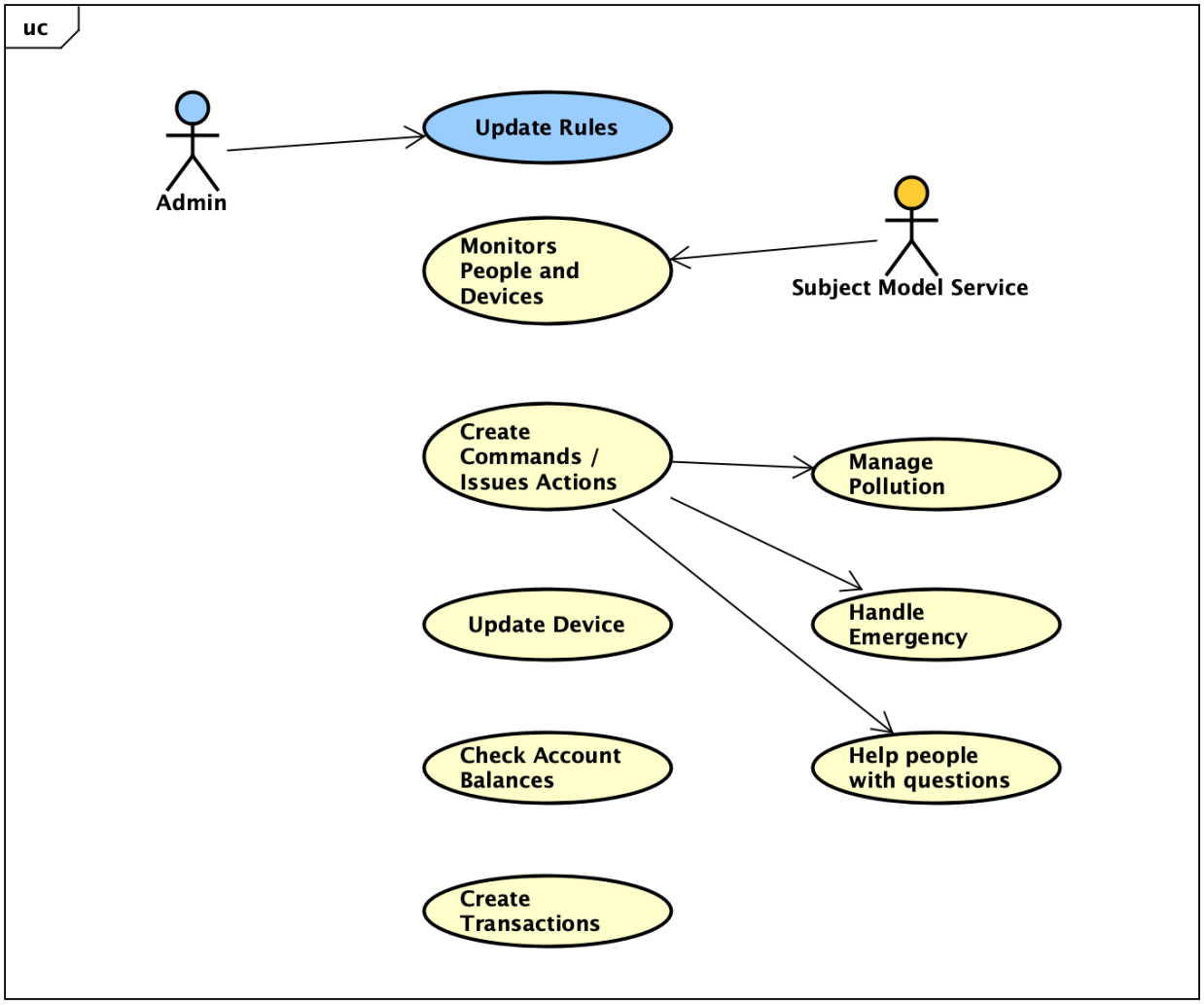
Name	Sensor or Appliance	Stimulus (within the context of a city <city_id>)	Action
Litter Event	Camera	Person <person_id> throws garbage on ground at lat <lat> long <long>	Speaker: “Please do not litter” Robot: "clean garbage at lat <lat> long <long>" Charge person <person_id> 50 units for littering.
Broken glass	Microphone	sound of breaking glass at lat <lat> long <long>	Robot: "clean up broken glass at lat <lat> long <long>"
Person Seen	Camera	Person <person_id> seen at lat <lat> long <long>	Update person <person_id> location lat <lat> long <long>
Missing child	Microphone	can you help me find my child <person_id>?	locate person <person_id> speaker: “person <person_id> is at lat <lat> long <long>, a robot is retrieving now, stay where you are." Robot: “retrieve person <person_id> and bring to lat <lat> long <long> of microphone.

Name	Sensor or Appliance	Stimulus (within the context of a city <city_id>)	Action
Parking Event	Parking Meter	Vehicle <vehicle_id> parked for 1 hour.	Charge the vehicle account for parking for 1 hour.
Bus Route	Microphone	Person <person_id> says, "Does this bus go to central square?"	Bus speaker: “Yes, this bus goes to Central Square.”
Board Bus	Camera	Person <person_id> boards bus.	Bus Speaker: “hello, good to see you <person_id>” If the person is a resident and has a positive account balance, charge persons account for the rate of the bus.
Movie Info	Kiosk	Person <person_id> says, “what movies are showing tonight?”	Speaker: “Casablanca is showing at 9 pm ” Display: “https://en.wikipedia.org/wiki/Casablanca_(film)#/media/File:Casablanca Poster-Gold.jpg”

Name	Sensor or Appliance	Stimulus (within the context of a city <city_id>)	Action
Movie Reservation	Kiosk	Person <person_id> says, “reserve 2 seats for the 9 pm showing of Casablanca.”	1. Lookup Person <person_id> 2. Check for positive account balance 3. If the person is a resident and has a positive account balance, charge the person 10 units 4. Speaker: “your seats are reserved; please arrive a few minutes early.”

Use Cases

The following use case diagram documents the high-level use cases supported by the Smart City Controller System.



Caption: UML Use case diagram of the controller.

There are 3 types of actors:

- Admin
- Subject Model Service
- Skynet

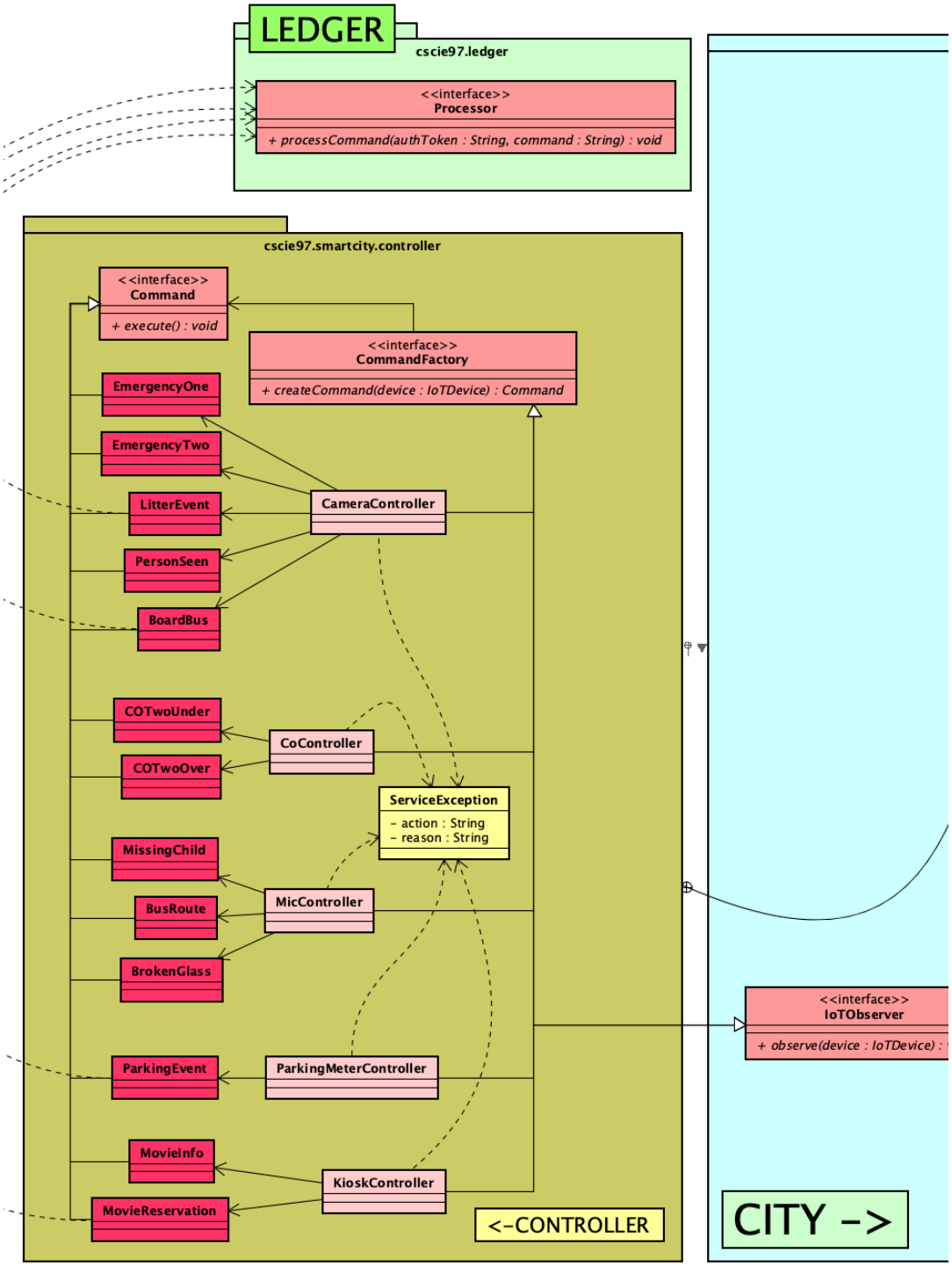
City Administrators are humans responsible for configuring the smart city controller. This includes defining the specific conditions to observe, and how to respond to them.

Subject Model Service monitors people and devices, and acts appropriately to them

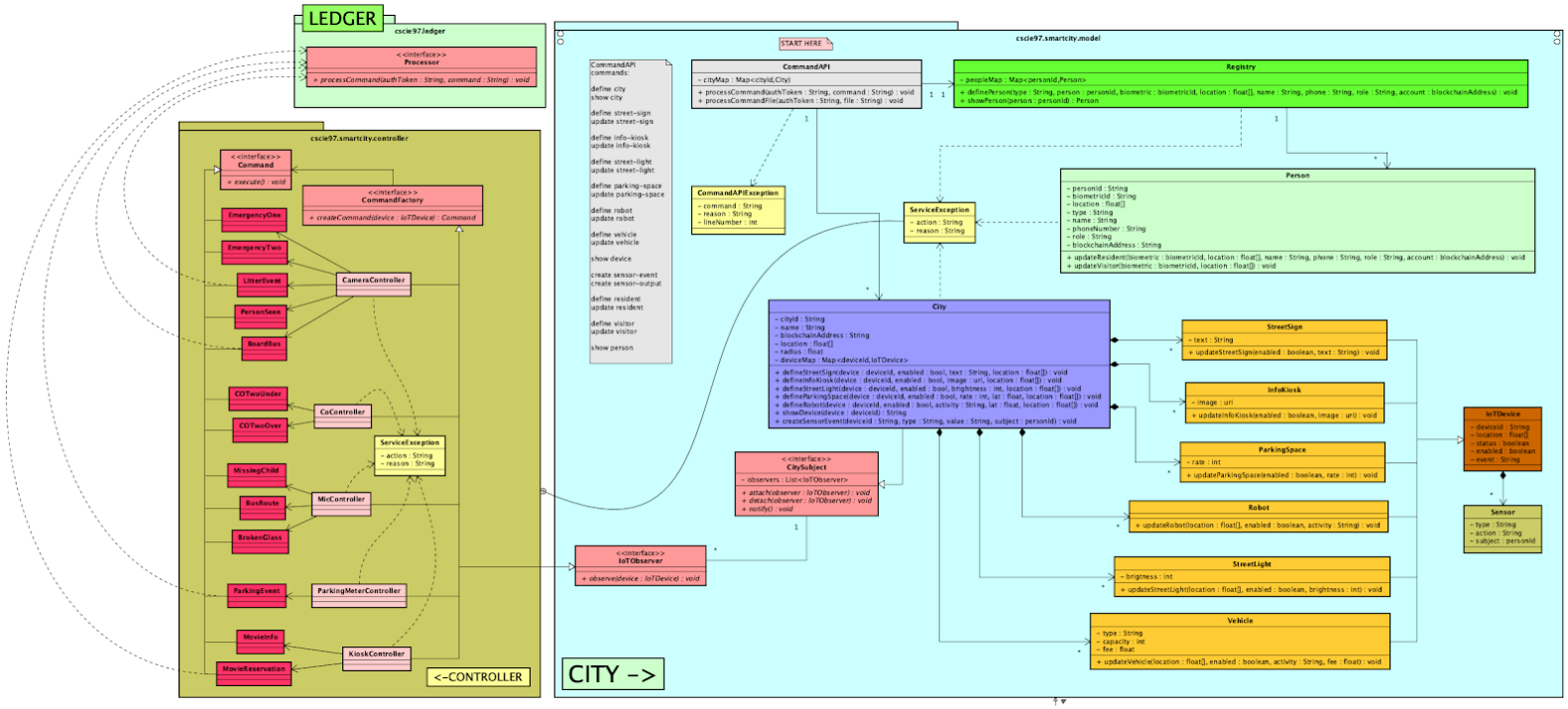
Implementation

Class Diagram

The following class diagram defines the Ledger implementation classes contained within the package "cscie97.smartcity.controller".



Caption: Smart City Controller Class Diagram



Caption: Entire Smart City System Class Diagram

Class and Interface Dictionary

CitySubject Interface (Class: City)

The city subject is an extension of the city class, which adds three new methods to the existing class, so that it can work as part of the OBSERVER pattern. These new methods follow the OBSERVER PATTERN.

Property Name	Type	Description
observers	List<IoTObserver>	A list of observers on the city

Method Name	Signature	Description
attach	(observer: IoTObserver):void	Add an observer to the list of observers
detach	(observer: IoTObserver):void	Remove an observer from the list of observers
notify	():void	Notifies all observers in the list of an update

IoTObserver Interface (Class: CameraController, CoController, MicController, ParkingMeterController, KioskController)

The IoT Observer are individual components of the controller that monitor for specific events in the city. The 5 controller classes all have this interface built in. These classes follow the OBSERVER PATTERN

Method Name	Signature	Description
observe	(deviceList : List<IoTDevices>):void	Observe the list of devices for specific events, per the requirements section

CommandFactory Interface (Class: CameraController, CoController, MicController, ParkingMeterController, KioskController)

The command factory builds commands based on the event it is responding to. The 5 controller classes all have this interface built in. These classes follow the COMMAND PATTERN.

Method Name	Signature	Description
createCommand	(deviceList : List<IoTDevices>):Command	Create a command to execute based on the info received, per the requirements section

Command Interface

The command are individual classes that are custom built by the command factories to perform a specific action. These classes follow the COMMAND PATTERN. This interface applies to these specific classes:

1. EmergencyOne
2. EmergencyTwo
3. LitterEvent
4. PersonSeen
5. BoardBus
6. COTwoUnder
7. COTwoOver
8. MissingChild
9. BusRoute
10. BrokenGlass
11. ParkingEvent
12. MovieInfo
13. MovieReservation

Method Name	Signature	Description
execute	():void	Execute the purpose-built command, per the requirements section

Command Classes

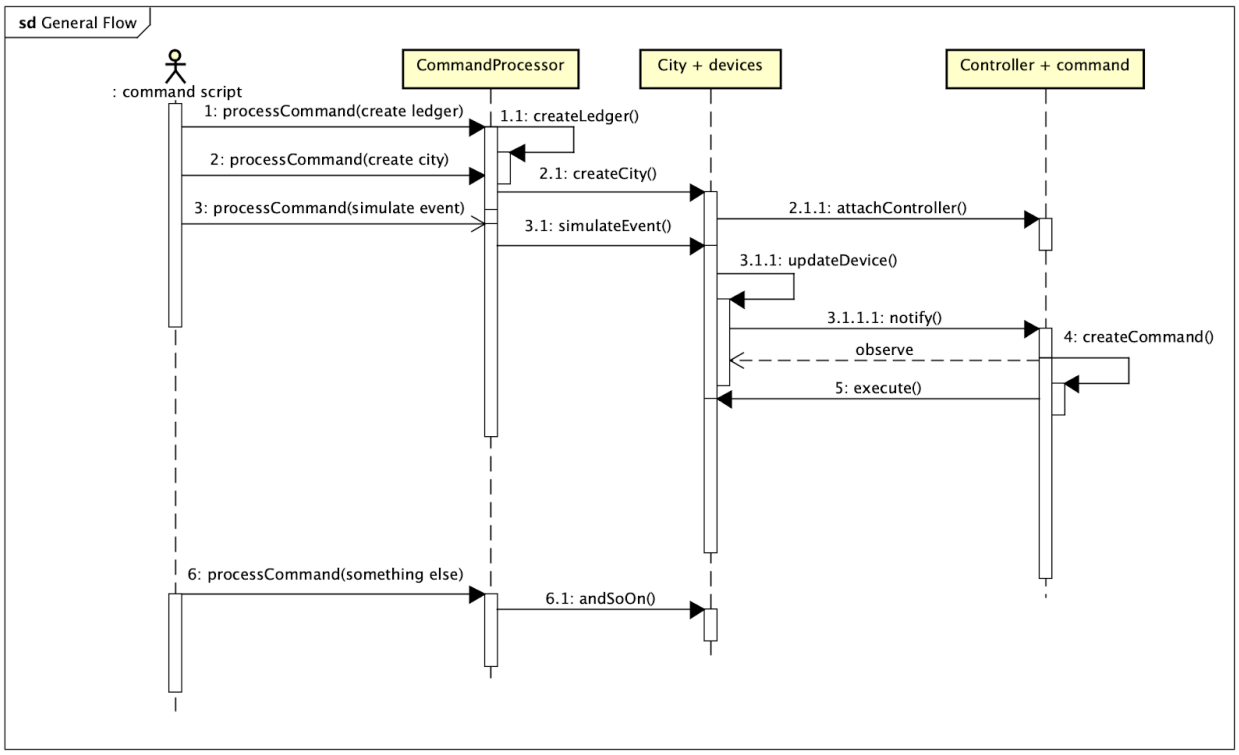
The general steps of action in each command class are summarized below

Name	execute() action
EmergencyOne	Report status of observing device Send half of all robots to emergency scene Send other half to help find shelter Report status of robots Report updated status of observing device
EmergencyTwo	Report status of observing device Send 2 robots to emergency scene Report status of robots Report updated status of observing device
LitterEvent	Report status of observing device Send 1 robots to clean litter Charge litter fee to person Report updated ledger balances Report updated status of observing device
PersonSeen	Report status of observing device Update location of person in the registry Report updated status of observing device
BoardBus	Report status of observing device Greet person Charge fare to person Report updated status of observing device
COTwoUnder	Report status of observing device Enable all cars Report updated status of observing device
COTwoOver	Report status of observing device Disable all cars Report updated status of observing device
MissingChild	Report status of observing device Find child location in registry Send robot to retrieve child Report updated status of observing device

BusRoute	Report status of observing device Answer question from person Report updated status of observing device
BrokenGlass	Report status of observing device Send robot to clean up Report updated status of observing device
ParkingEvent	Report status of observing device Charge car parking fee Report updated status of observing device
MovieInfo	Report status of observing device Answer request from person Report updated status of observing device
MovieReservation	Report status of observing device Charge person for tickets Notify person of reservation Report updated status of observing device

Implementation Details

Below is a sequence diagram demonstrating the general implementation:

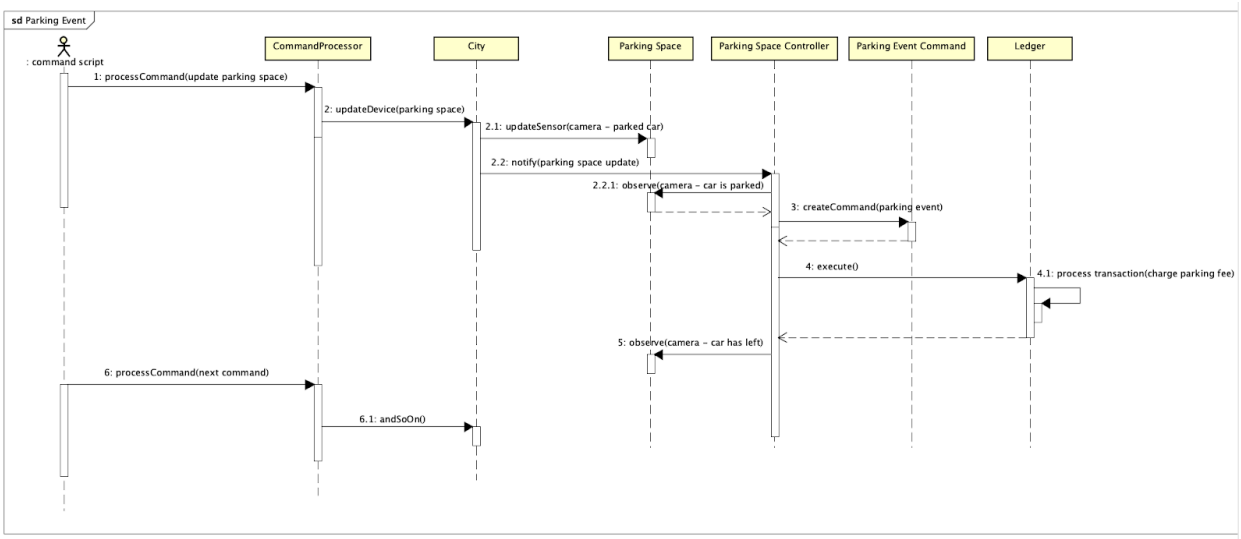


Caption: Sequence Diagram, general flow

In the general flow diagram, the action starts with a command script input. This input is first read by the command processor, which then is passed on to other classes, depending on the command. The first few commands would be to create the ledger to hold transactions, then create a city and devices. In creating the city, it will create new controllers and attach them as OBSERVERS. Afterward, the city is ready to process more actions from the command processor.

The command processor can then take a simulated device event command, and pass it to the newly created city, which may then update the device. If it does, the city will notify the controller of the update. The controller, acting as an OBSERVER, scans through the info that was provided in the notification, and creates a command, based on the COMMAND pattern, and then executes the command. This execution may cause more updates to the devices.

A more detailed sequence diagram of a parking event is shown below:



Caption: Sequence Diagram, parking event, detailed

This sequence above is specific to the parking event. The command script provides a command to simulate a parking event, which is to update the parking space camera to show a parked car. The command processor reads the command, and passes on the work to the city, which goes and finds the parking space, updates the camera, and then notifies the controller of the update. The controller takes a look at the camera, sees a parked car, and creates the COMMAND specifically made for parking events. Once the controller receives the specifically made COMMAND, it gets executed, which opens the ledger to charge for the parking. It then checks the camera again, and finds the car has left, so it will not charge it again. The command processor is then ready to process another command.

Exception Handling

ServiceException

The ServiceException is returned from the methods in response to a specific error in observing an event or creating a command. It captures the action that was attempted and the reason for the failure. ControllerException should be caught by the top level CommandAPIException from the City package. It is the same ServiceException in the city package.

Property Name	Type	Description
action	string	action that was performed (e.g., “emergency command”)
reason	string	Reason for the exception (e.g. “printer out of paper”).

Testing

Implement a similar TestDriver class based on the city model service, that implements a main() method. The main() method can call these methods:

CommandAPI.processCommandFile(authToken: String, file:String)
CommandAPI.processCommandFile(authToken: String)

The main method can receive a command file, or allow a user to enter commands in a console if no file is specified. The TestDriver class should be defined within the package “cscie97.smartcity.test”.

- The program and tests should compile with this command executed in the base directory:

javac cscie97/ledger/*.java cscie97/smartcity/model/*.java cscie97/smartcity/controller/*.java cscie97/smartcity/helper/*.java cscie97/smartcity/test/*.java

- The test should run with the command:

java -cp . cscie97.smartcity.test.TestDriver NAME_OF_CITY_SCRIPT

- Or, if a separate set of ledger commands are provided:

java -cp . cscie97.smartcity.test.TestDriver NAME_OF_LEDGER_SCRIPT NAME_OF_CITY_SCRIPT

Risks

Since this is a system dealing with people and devices, it will be subject to hackers who will attempt to undermine it. City administrators are required to have an authorization token to use the city service. All rules implemented to the controller should be carefully reviewed from a high level to ensure they do not result in unintended consequences, for example: the rule to disable all vehicles can block any fire rescue attempts as no rescuers can arrive at the scene in time; asking for movie info can result in the speakers stopping broadcast of an emergency; disabled vehicles telling riders that it will go to Central square, but will actually trap riders in CO2 poisoning; sending robots to help people find shelter can leave the city defenseless from an alien invasion.