Smart City Controller Service Design Document

Date: 10/17/2020
Author: Mariam Gogia
Reviewer(s): Eric Moon, Kevin Sun

Table of Contents

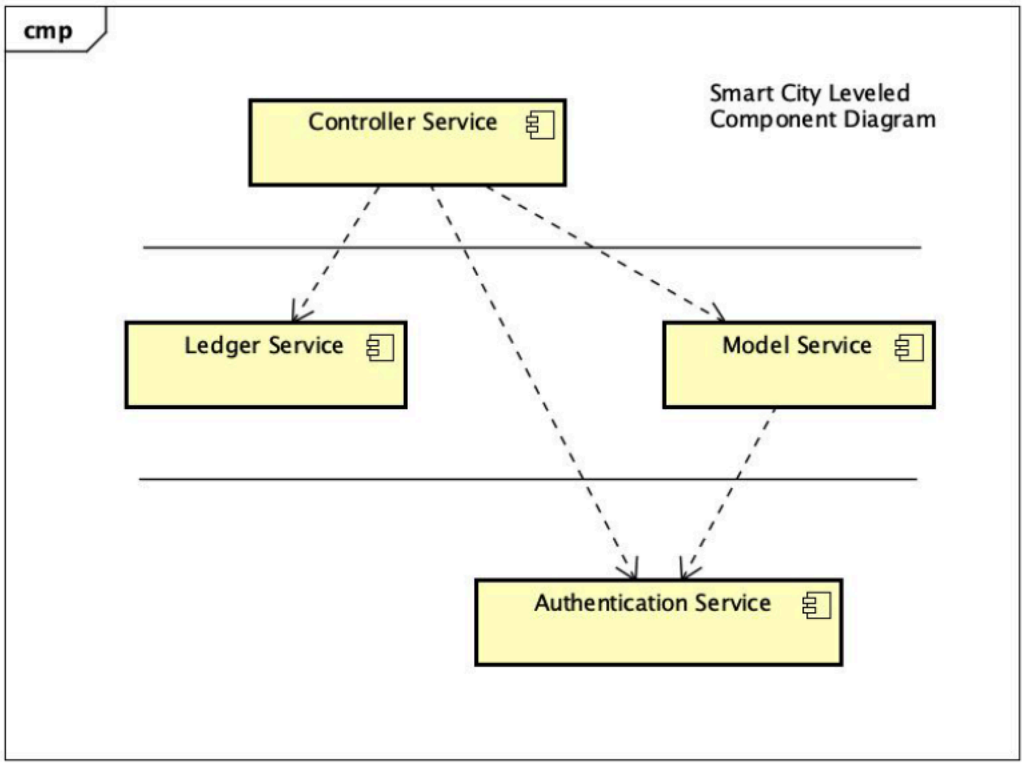
Introduction

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

Overview

The Controller Service represents one of the 4 modules of the Smart City system. This pillar is responsible for monitoring devices in the city through their 4 built-in sensors. The controller gathers information about the city from devices' microphones, cameras, CO2 meters, and thermometers, and issues action commands based on the predetermined rules. The controller also monitors people in the city to provide appropriate service to them through IoT devices.

The Controller "ties" all four components of the system together as it has knowledge of all of them and monitors and controls the actions of all city objects (devices, people), and executes transactions through ledger service.



Requirements

This section provides a summary of the requirements for the Controller Service.

The Smart City Controller Service should fulfill the following requirements:

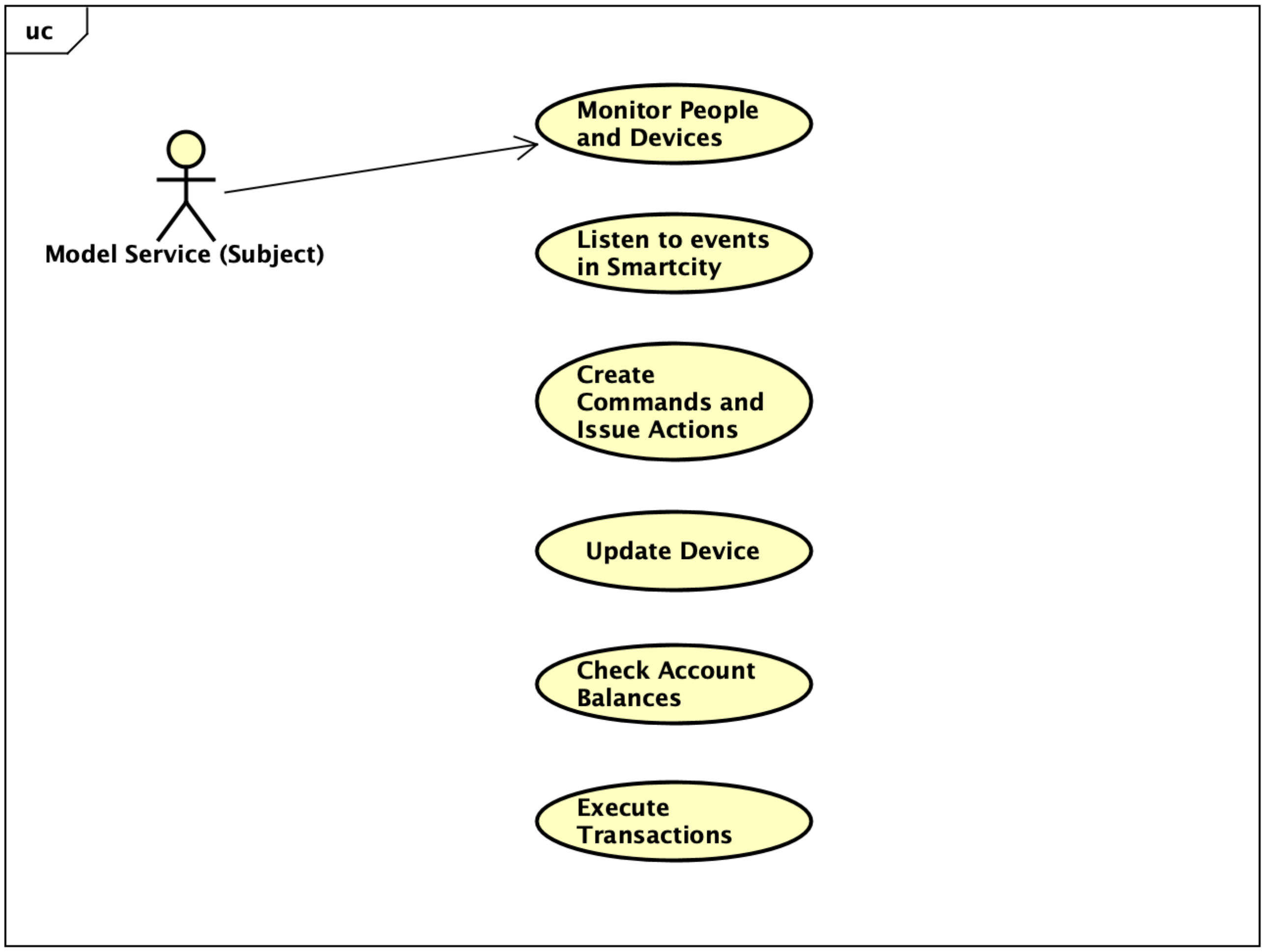
- Monitor the events received from device sensors
- Respond to sensor events via issuing action commands based on rules.
- Listen to voice command coming from IoTs microphone and respond by issuing an appropriate control/action command to that IoT
- Process financial transactions via Ledger Service using Unit currency
- Responsible for maintaining the safety in the city
- Log all commands, rule executions, and resulting actions.

Design Requirements

- Use the Observer Pattern to let the Smart City Controller be the observer of the events taking place in its subject - Smart City Model Service sensors.
- Use the Command Pattern for the actions performed by the Smart City Controller Service
- Use the Ledger Service Module to handle transactions and to check balances.

Use Cases

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


uc
Model Service (Subject)
Monitor People and Devices
Listen to events in Smartcity
Create Commands and Issue Actions
Update Device
Check Account Balances
Execute Transactions

Actors:

Model Service (Subject):

As controller service is more of ‘internal’ service, it has only one outside actor a Model Service, which is a subject it is listening to as that is where the sensor events take place. Model Service receives events from the simulator and sends it over to the Controller.

Use Cases:

Monitor People and Devices:

Controller monitors location of the devices and people within the city as well as the status of the devices.

Listen to the events in SmartCity:

The Controller Service is responsible for receiving and acknowledging the events emitted from the device sensors.

Create Commands and Issue Actions:

In response to device sensor readings The Controller generates commands and issues actions applying the predetermined rules.

Update Device:

Update the device state based on an event or voice command. For example, change the display screen on kiosk.

Check account balances:

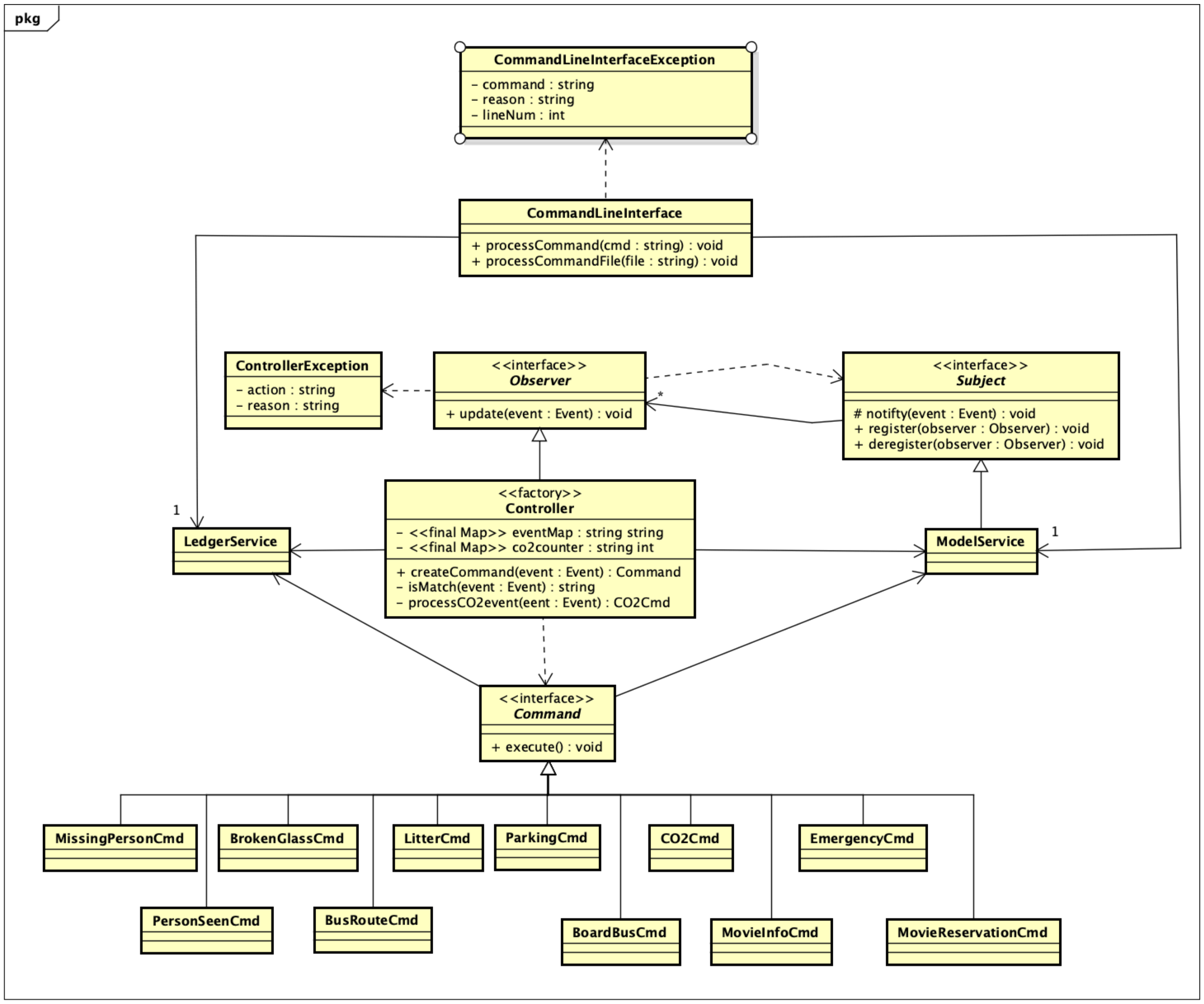
Before resident uses a paid service, Controller must check if the resident has a positive balance.

Execute Transactions:

If a resident uses paid service, Controller must charge the person using Ledger Service.

Implementation

Class Diagram


pkg
CommandLineInterfaceException
– command : string
– reason : string
– lineNum : int
CommandLineInterface
+ processCommand(cmd : string) : void
+ processCommandFile(file : string) : void
ControllerException
– action : string
– reason : string
<<interface>>
Observer
+ update(event : Event) : void
<<interface>>
Subject
# notifty(event : Event) : void
+ register(observer : Observer) : void
+ deregister(observer : Observer) : void
*
<<factory>>
Controller
– <<final Map>> eventMap : string string
– <<final Map>> co2counter : string int
+ createCommand(event : Event) : Command
– isMatch(event : Event) : string
– processCO2event(eent : Event) : CO2Cmd
1
LedgerService
ModelService
1
<<interface>>
Command
+ execute() : void
MissingPersonCmd
BrokenGlassCmd
LitterCmd
ParkingCmd
CO2Cmd
EmergencyCmd
PersonSeenCmd
BusRouteCmd
BoardBusCmd
MovieInfoCmd
MovieReservationCmd

Design Pattens

The implementation employs three types of design patterns to the problem.

1. <u>Observer Pattern</u>: Controller Service is the observer to its subject - Model Service. The Controller listens to sensor events emitted from devices in the Model Service and updates them accordingly. The Model Service has no knowledge about the Controller Service.

2. Factory Pattern: Controller Service, through ‘createCommand’ function, serves as a command factory. Based on the type of passed in event, it creates an object of an appropriate command class.
3. Command Pattern: The actions to be performed by the Smart City Controller follow the Command Pattern.
4. Singleton Pattern: has been applied to Ledger and Model service classed.

Interfaces & Abstract Classes

- Observer Interface - inherited by Controller Service factory to allow listening to changes in device sensors.
- Subject Interface - inherited by Model Service to allow registration, notification, and undo registration functionalities for the Controller (observer)
- Command - inherited by concrete command classes. There is one command class per rule. Their implementation of ‘execute()’ implements the actions to be carried out as per given rules.

Dependencies & Associations

Dependencies:

- Observer interface depends on a Subject interface - Controller depends on Model Service
- Controller Service depends on a command interface, it cannot carry out its “duties” without using command interface and its concrete classes.
- ControllerException and CommandLineInterfaceException depend on Controller and and CommandLine, respectively.

Associations:

- Model Service - Controller Service: one-to-many relationship - subject has (can have) many observers.
- Controller Service and the Command interface, both have associations with Ledger and Model services. They must have access to singletons of these classes to carry out their functionalities.
- Command Line - initializes singletons of Ledger and Model services. It also defines objects in model service and creates accounts and transactions in ledger service.

Other Comments: LedgerService and ModelService on the diagram are placeholders for respective modules. Class diagrams of those modules are hidden for readability.

Class Dictionary

Model Service (modifications)

Several modifications were made to Smart City Model Service implementation. As per requirement, Observer pattern was applied and Model Service now inherits from the Subject interface and therefore has 3 additional functions. Functions are described below:

Method Name	***Signature***	***Description***
register	(observer:Observer):void	Adds the observer to observers list
unregister	(observer:Observer):void	Removes the observer from observers list
notify	(event:Event):void	Function is placed in createSensorEvent, it is to notify the observer that the new sensor event came in

The Notify function helps fulfill the requirement of monitoring events and listening to microphone inputs by notifying the observer of incoming events.

Another modification in Smart City Model Service implementation is in its event class. Previously an event class had only 3 attributes: sensor type, subject, and action. Now, it has the IoT property to store the IoT object the event is coming from. Singleton pattern has been applied to model city to use its one instance throughout.

To fulfill logging requirement, a function createSensorOutput was added to the Model.

Method Name	***Signature***	***Description***
createSensorOutput	(str: string):void	Prints the details of the response to the event.

<<factory>> ControllerService

Controller Service is a factory that creates appropriate commands based on the received event type. It implements from the Observer interface and listens to the Subject - Model service. It contains the update method which by calling the appropriate execute method, makes the system react appropriately to incoming events.

*** Note: For simplicity, not all private maps were shown on a class diagram.*

Property Name	***Type***	***Description***
eventMap	Map <String: String>	Map contains the strings that trigger rules. Strings are mapped to appropriate command class names. e.g. “fire” : “EmergencyCmd.” The map gets preloaded when controller is initialized.
co2Counter	Map <String:Integer>	Map counts if 3 unique devices reported co2 levels > 1000. e.g. If such thing happens in city_1, the map will contain the key:value pair city_1:1, when 3 unique devices report co2 level < 1000, the same pair becomes city_1:0. The map is used to ensure that enabling cars event gets triggered only after disabling cars event has occurred.
co2Above	Map <String, HashSet<String>>	Keeps track of device IDs that reported co2 value > 1000 in specific city. Using hash set ensures that no duplicates end up in the set, so it only records reporting from 3 unique devices in the city. Example: city_1:city_1:bot_1

co2Below	Map <String, HashSet<String>>	Keeps track of device IDs that reported co2 value < 1000 in specific city. Using hash set ensures that no duplicates end up in the set, so it only records reporting from 3 unique devices in the city. Example: city_1:city_1:bot_1

Method Name	***Signature***	***Description***
update	(event:Event):void	Calls createCommand to create a command of an appropriate class given the event. If command is not null, it calls the execute method on it. Only exception is the event coming from co2 meter, in which case update calls 'processCO2event' instead of createCommand.

isMatch	(event:Event):String	Receives the event as an argument, extracts the event value (action) and uses it as map key to look for a match in eventMap. If the match is found, the corresponding value of the key is returned, if not the empty string is returned. Because the missing person's ID or parking car's id can vary, the event values for MissingPersonCmd and ParkingCmd are not fixed. Therefore, these commands are not loaded in map, instead isMatch checks if value string contains part of the sentence e.g. "can you help me find my child" in missing person's case and "parked" in parking command case and if so, returns the strings representing an appropriate command class names.
createCommand	(cmd: String, event:Event):Command	createCommand receives a string cmd returned by isMatch. Cmd contains the class name - instance of which to be created, e.g. "EmergencyCmd." createCommand first appends the package name in front of the cmd value and then looks for a command class with matching name. Finds it and creates an instance, passing the event in the constructor. All command classes take in event as their constructors.

processCO2event	(event:Event):CO2Cmd	Each time the event is received from co2 sensors, processCO2event gets called. It counts if 3 unique devices consecutively reported levels higher than 1000 or lower than 1000. If 3 unique devices report levels higher than 1000, CO2Cmd instance with int 0 parameter gets created, 0 indicates to disable car. If after disabling cars 3 unique device reports levels below 1000, CO2Cmd with int 1 parameter gets created. 1 indicates to enable the cars. processCO2 returns CO2Cmd back to update where update calls the execute method on it.

Command Concrete Classes

All concrete classes inherit from Command interface.

As per requirement, Command Pattern is used to implement the rules.

Each Command Class has a private field event, representing the incoming event.

Through execute() command, the system fulfills the requirement of appropriately responding to the incoming event and in some cases checking balances and processing transactions.

EmergencyCmd - Class manages rules Emergency 1 and Emergency 2. Its execute method is the implementation of actions outlined in rules.

Method Name	***Signature***	***Description***

execute	():void	If the type is fire, flood, earthquake or severe weather, announce it in the city, send 1/2 of robots at the location, make the rest help people with finding the shelter. If the type is “traffic_accident” announce to stay calm, calculate the distances between the accident and robots, find 2 nearest robots and send them to the location If there are no robots throw ControllerException

CO2Cmd - Class manages rules for CO2 meter events.

Method Name	***Signature***	***Description***
execute	():void	Look for all vehicles that are type car and are assigned to the city where the co2 values have been reported. If enableCars is 1, set all cars in the city to enabled, if enabledCars is 0, disable all cars.

LitterCmd - Class manages the rule for littering the city.

Property Name	***Type***	***Description***
littering_fee	int	Final variable set to 50. Fee charged for littering

Method Name	***Signature***	***Description***
execute	():void	Look for a robot and send one to clean up the garbage. Robot speaker:"please do not litter". Look for a person who littered, if the person is resident find his/her ledger account, check if they have sufficient funds and charge 50 units plus 10 units transaction fee.

BrokenGlassCmd - Class manages the rule for broken glass in the city

Method Name	***Signature***	***Description***
execute	():void	Find enabled robot and send to clean up the glass.

PersonSeenCmd - Class manages the rule for seeing the person in the city

Method Name	***Signature***	***Description***
execute	():void	Person detected at certain location —> update his/her location to that location

MissingPersonCmd - Class manages the rule for missing (child) a person in the city.

Method Name	***Signature***	***Description***
execute	():void	Retrieve the id of the missing person, confirm such person exists and determine his/her location. Send a robot to bring the person to the location of a searcher.

ParkingCmd - Class manages the rule for parking in the city

Method Name	***Signature***	***Description***
execute	():void	Retrieve the id of the parking vehicle, check if it has sufficient account balance to cover the hour charge. If so, charge the vehicle hourly rate plus transaction fee and set the parking space unavailable.

BusRouteCmd - Class manages the rule for answering residents questions about bus route

Method Name	***Signature***	***Description***
execute	():void	Bus speaker output: “Yes, this bus goes to Central square”

BoardBusCmd - Class manages the rule for residents and visitors boarding the bus

Method Name	***Signature***	***Description***
execute	():void	Bus welcomes the person by his/her name. Checks if the person is a resident and has sufficient balance, if so, charges the bus fee plus the transaction fee.

MovieInfoCmd - Class manages the rule for giving movie information to people

Method Name	***Signature***	***Description***
execute	():void	Information Kiosk’s speaker outputs: “Casablanca is showing at 9pm” and changes its display to movie link.

MovieReservationCmd - Class manages the rule for making a movie reservation via information kiosk

Property Name	***Type***	***Description***
movie_fee	int	Movie ticket cost, final attribute set to 10 units.

Method Name	***Signature***	***Description***
execute	():void	Check if the person is a resident and has sufficient funds, if so, charge 10 units + transaction fee. Announce the outcome

CommandProcessor
Processes the script

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

CommandProcessorException
Invoked when there are errors in the script

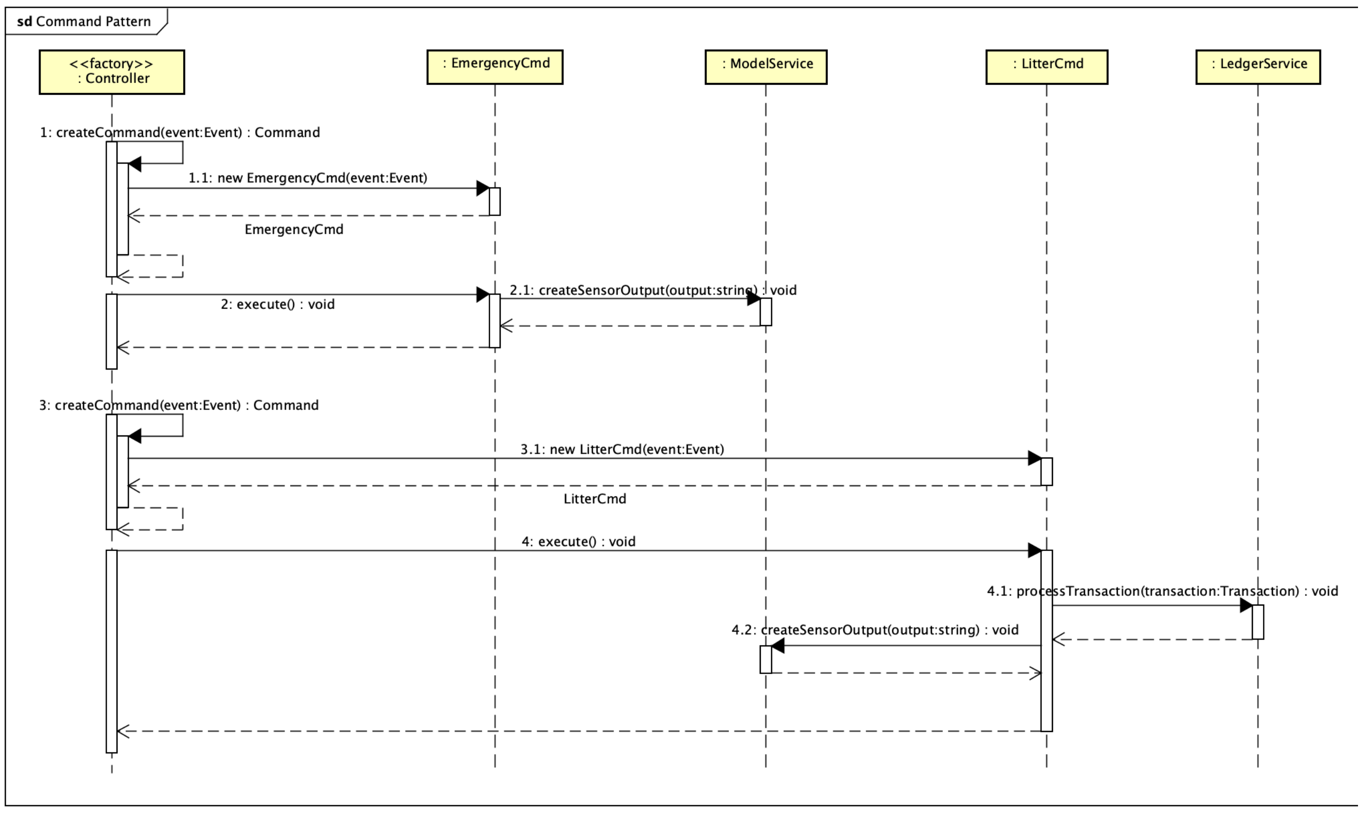
Property Name	***Type***	***Description***
action	string	Performed action (command)
reason	string	The reason it failed (e.g. invalid number of arguments)
lineNum	int	The line number that caused the error

ControllerException
Invoked when errors occur in Controller Module

Property Name	***Type***	***Description***
action	string	Performed action (command)
reason	string	The reason it failed (e.g. no such person)

Implementation Details

Sequence Diagrams

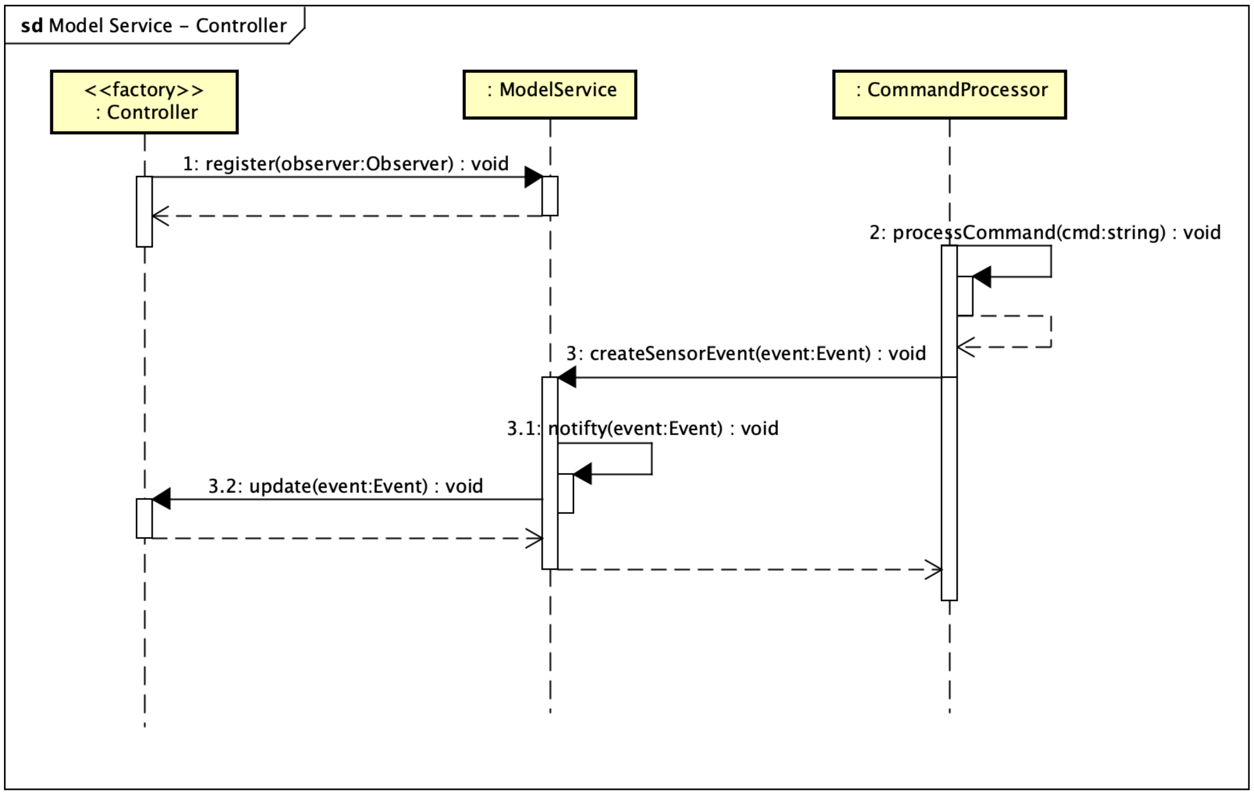

sd Command Pattern
<<factory>> : Controller
: EmergencyCmd
: ModelService
: LitterCmd
: LedgerService
1: createCommand(event:Event) : Command
1.1: new EmergencyCmd(event:Event)
EmergencyCmd
2: execute() : void
2.1: createSensorOutput(output:string) : void
3: createCommand(event:Event) : Command
3.1: new LitterCmd(event:Event)
LitterCmd
4: execute() : void
4.1: processTransaction(transaction:Transaction) : void
4.2: createSensorOutput(output:string) : void

Command Pattern Sequence Diagram:

(1) createCommand(event: Event) method gets called in Controller class, it determines that the concrete class to be created is (1.1) the EmergencyCmd, after determining the concrete class of the Command, the (2) execute method in EmergencyCmd class is

invoked. Execute method fulfills the requirements of the rules for incoming emergency command and sends the results to Model Service (2.1) createSensorOutput method to output the results of the executed rules.

Another event is received from the simulator, Controller factory issues (3) createCommand(event: Event) to determine the concrete class of the command. It gets back a new instance of (3.1) LitterCmd, after which the (4) execute method in LitterCmd class is invoked. Execute method fulfills the requirements of the rules, it checks if the littering person is a resident and has sufficient funds, if so, it charges the resident for littering by invoking (4.1) processTransaction in Ledger Service. After charging is complete, it calls (4.2) createSensorOutput in Model Service to output the result of the executed rules.


sd Model Service – Controller
<<factory>> : Controller
: ModelService
: CommandProcessor
1: register(observer:Observer) : void
2: processCommand(cmd:string) : void
3: createSensorEvent(event:Event) : void
3.1: notifty(event:Event) : void
3.2: update(event:Event) : void

Observer Pattern Sequence Diagram:

Using (1)register method, the observer (Controller) registers itself to Model Service (Subject). (2)Command processor (simulator) reads script and processes create sensor-event command. Calling this command creates the event object and calls Model Service's (3)createSensorEvent event function to register the event in the system.

Model Service does validation and error checking after which it calls (3.1) notify function to notify the observer of the new event. Notify calls the (3.2)update function in the controller which figures out how to proceed with the incoming event.

Other Implementation Details and Comments

Patterns

Command Pattern and Observer Pattern were used as per requirements. However, besides the above-mentioned patterns, the implementation also applies factory and singleton patterns. The controller is an Observer, as well as a factory for commands. Its createCommand along with isMatch function determines which concrete class command belongs to. Ledger and Model services were turned in singletons.

Program Initialization Details

One of the suggested ways to initialize the system was to run ledger and smart city scripts separately. I decided not to follow this approach for several reasons. Initializing Ledger with the script means that one has to know beforehand what objects will be in the city. For example, one cannot dynamically add define-city or define-resident in smart city script because such objects will not have ledger accounts and therefore cannot participate in city activities. Instead, I decided to dynamically create accounts transactions city objects. For example, if the command define-city is received, along with defining the city, the CommandLineInterface will also define the ledger account for it. The same is true for IoTs and residents. When residents' accounts get created, I process a funding transaction for them so they have sufficient funds to participate in the simulation. I also fund vehicle's accounts because they need money for parking events. In conclusion, CommandLineInterface reads the smart_city script and dynamically creates ledger accounts and funding transactions. The initialized Ledger service's credentials are: name: "ledger" description: "smart city ledger service" seed: "1"

Ledger Account Funding & Charging

No approach has been provided as to how to fund IoTs and residents, however, to fulfill the requirements of the rules, sufficiently funded accounts are needed. Upon account creation, all residents and vehicles get 1000 units from the master's account. All other

IoTs, including the cities, will have their own unique Ledger accounts, however, with 0 funds.

Every transaction fee is 10 units. Movie reservations, littering, parking, or riding a bus will cost the fee plus a 10 unit transaction fee, fully incurred by the payer account. In case of insufficient funds (calculation of which considers transaction fee as well) event rule will not get executed, CommandException will be thrown and caught.

Other Comments

'Missing Child' rule has been generalized and turned into 'Missing Person,' this rule allows finding not only children but other missing people in the city too.

In hindsight, it would have been more practical while designing the Model Service to have kept a separate map for robots in the city. Controller rules heavily rely on robots and each time the rule is invoked, current implementation requires to look up robots in a big map of IoTs. Also, it would have been more logical if vehicles "made their own money" by having a rule to give rides to residents and charge the fee. This way, they could pay for their parking without having to pre-fund their accounts. All of these can be implemented in coming sprint.

Exception Handling

The system as a whole handles 4 different types of exceptions and has an exception class for each type. These 4 exceptions are:

1. Command Processor Exception
2. Model Service Exception
3. Ledger Exception
4. Controller Exception

Command Processor Exception handles any file/script related issues. If there is a syntax error in the script or the system cannot read the file, command processor exception is invoked.

Model Service Exception is invoked when the system encounters issues in defining, querying, or updating the objects (people, cities, IoTs) in the city.

Ledger Exception handles issues related to invalid transaction requests, account querying/creation, and funding accounts.

Controller Exception is invoked where there are issues regarding objects to be updated. (e.g. “person does not exist”, “insufficient funds”).

Testing

The system is tested using 2 scripts. The first script defines and updates objects and tests basic functionalities of implemented rules. The second script tests edge cases and also confirms that rules were successfully invoked. Test file is documented and commands are headed with comments as to what to expect.

Script 2 Details:
Defining:
2 cities
4 robots per city
1 resident 1 visitor per city
2 cars per city
Other types of IoT devices - 1 per city

- Testing Rules for Emergency:
  - After invoking Emergency 1 (e.g. fire) show details of all 4 robots to see that 2 of them were sent to the emergency location and 2 are helping people find shelter.
  - After invoking Emergency 2 (traffic_accident), 2 nearest robots are deployed to the location and their device ids are printed. Using returned device ids, show details of those robots to ensure that rule was successfully executed.

- Testing Rules for CO2 levels
  - First, 3 unique devices report co2 levels below 1000. This should not invoke a rule, “below co2” rule is invoked only after “above co2” has been invoked.
  - Next, 3 non-unique devices report co2 levels above 1000. This also should not invoke a rule because the event is reported by non-unique devices.

  - Another device reports high co2 levels it makes the total of 3 consecutive unique devices reporting the high levels, so, the rule is invoked. To ensure that cars in the city are disabled, calling show device command on one of the cars in the city and one of the cars in another city (to ensure that cars were disabled in the correct city).
  - 3 unique devices in the city report low levels of co2, rule is invoked, cars are enabled. Show details of one of the cars to ensure that rule was correctly applied to cars.

- Testing Rules for Littering
  - Use a visitor as a subject of the littering event. Visitors cannot be charged as they do not have ledger accounts, the event response should say so. The response should also return the device id of the robot sent to clean the garbage.
  - Use returned device id to show details of the robot and ensure that it was indeed deployed to the location.
  - Use a resident as a subject of the littering event. Response event should the name of the resident and the fee amount charged.

- Testing Rules for Broken Glass
  - Event response to broken glass event contains the id of the robot that was sent to clean up the broken glass.
  - Use the device id to show details of the robot sent and ensure it was updated correctly.

- Testing Rules for Person Seen
  - After invoking Person Seen rule, show details of the person seen to ensure that his/her location was correctly updated.

- Testing Rules for Missing Person
  - After invoking Person Seen rule, show details of the missing person to ensure that his/her location was correctly updated to the location of the reporting device.
  - Try to request to find a person that does not exist - expect exception.

- Testing Rules for Parking Event
  - After invoking the rule, print the details of the car to see if its location, activity, and the latest event fields were correctly updated.

- Testing Rules for Boarding Bus

  - london:bus_2 fee was intentionally set to 2000 so that “insufficient funds” exception would be invoked if one of the residents tried to bored the bus.
  - When visitor boards the same bus he/she is welcomed with the message “Enjoy your free ride”

- Testing for Movie Info Rules
  - After invoking the rule, show details of info kiosk and observe if the display and the latest event field were correctly updated.

- Testing for Movie Reservation Rules
  - Test that visitors cannot make movie reservations as they do not have ledger accounts.
  - Show how residents can make successful reservation if they have sufficient funds.

Risks

The system has several shortcomings.

The pipeline can be overwhelmed if a lot of devices simultaneously report events. Also, currently, there is no mechanism to prevent one rule from overriding another. For example, if one event sends a specific robot (robot 1) to do something and then immediately another event requests robot 1 to do something else, there is no mechanism to prevent aborting action or to confirm that the first action was successfully fulfilled and now the robot is free to do something else.

IoTs are not intelligent, they can only respond to predetermined queries. The smart city will require IoTs with AI - trained and sophisticated.

The system needs a more secure way to deal with the resident’s ledger accounts. Model service, where the definition of objects happens, does not communicate to ledger service, this disconnection makes the opportunity for the following scenario: when a person is defined in model service, he/she is given an account address based on which the ledger account for this person is created. One can define another person and give him/her the same account address, define-person command will call ledger’s ‘create account’ function, however, the ledger will check and see that such an account already exists and will not create a new one. Not being able to create a new account, does not prevent the definition of a new person, so two people in the system will be assigned 1 ledger account, this allows a second person to fraudulently use the other person's account address. I strongly believe that the Smart City Model Service needs to have knowledge of the accounts list at a minimum.