Controller Service Design Document

Date: 10/26/2020
Author: Christopher Jones
Reviewer(s):

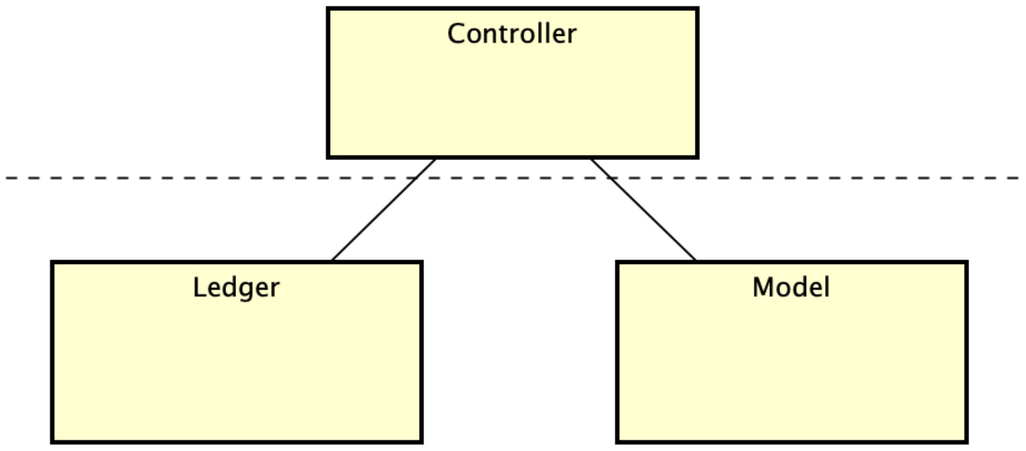
- Muhammad Ayub

Introduction

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

Overview

The Smart City Controller Service is responsible for monitoring the people and devices within a given city and regulating their activity within their respective city. It responds to status updates and actions that occur within the city through the lens of the device sensors. Additionally, it assures that the people within these cities are living as they should by assuring that they are financially stable, acknowledged for hindrances within the city, notified for emergencies within the city, and have their general questions and concerns responded to.



Requirements

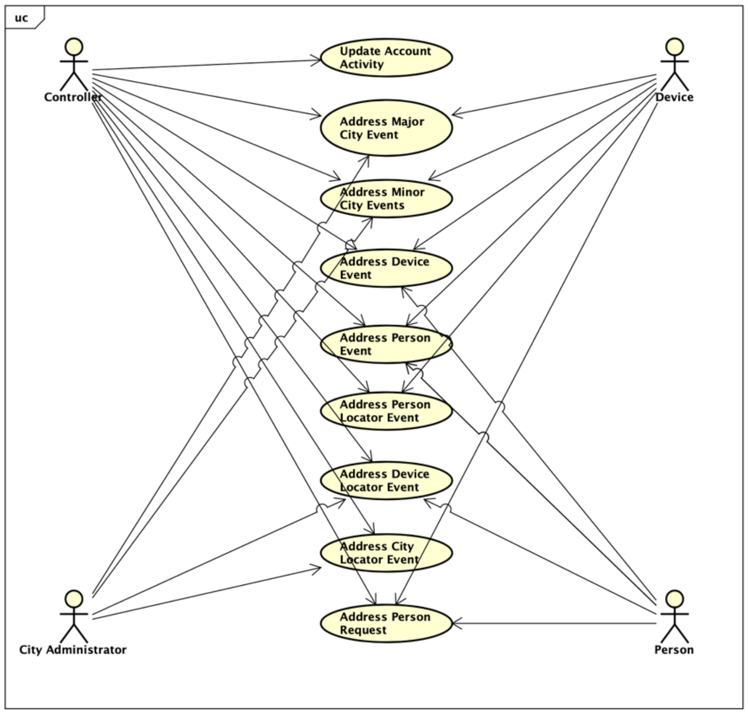
The section proves a summary of the requirements for the Controller Service.

The Smart City Controller Service is primarily responsible for monitoring the Model and Ledger services and delegating tasks out to the devices or assisting the people within it. These services can be considered Observable provided that they allow the Controller Service to subscribe to them. This subscription can be started and ended at any time. Once subscribed, the Controller can freely get updates from these observables and propagate commands back to them if needed.
The Controller Service is made up of the following elements:

- Interfaces
  - Observer
  - Commands
- Tracking
  - Subscriptions
  - Event Queues
  - Observed Events
  - Observable Accounts
- Observable services
  - Model Service

Use Cases

The following Use Case descripts the use cases supported by the Controller Service.



There are 4 types of actors to consider:

- Controller

- Administrator
- Person
- Device

Controllers are the main monitors and managers of the Smart Cities. They regularly monitor and assist devices that have successfully attached to the its subscription list. The controller allows of any device to freely attach, detach, or notify it their events. Additionally, the controller can deploy city devices to assist the visitors and residents within it. It also manages the interactions between the global, city, and individual user accounts and assures that they are up to date as well.

Administrators are the responders of large city-wide activities. They allow broadcasts to circulate through the entire city through each of its various devices. It additionally identifies its individual residents and visitors and utilizes its devices to assist them during major events and emergencies.

People are the individual visitors and residents within any given city. They both interact with, and are guided by, the devices of the city.

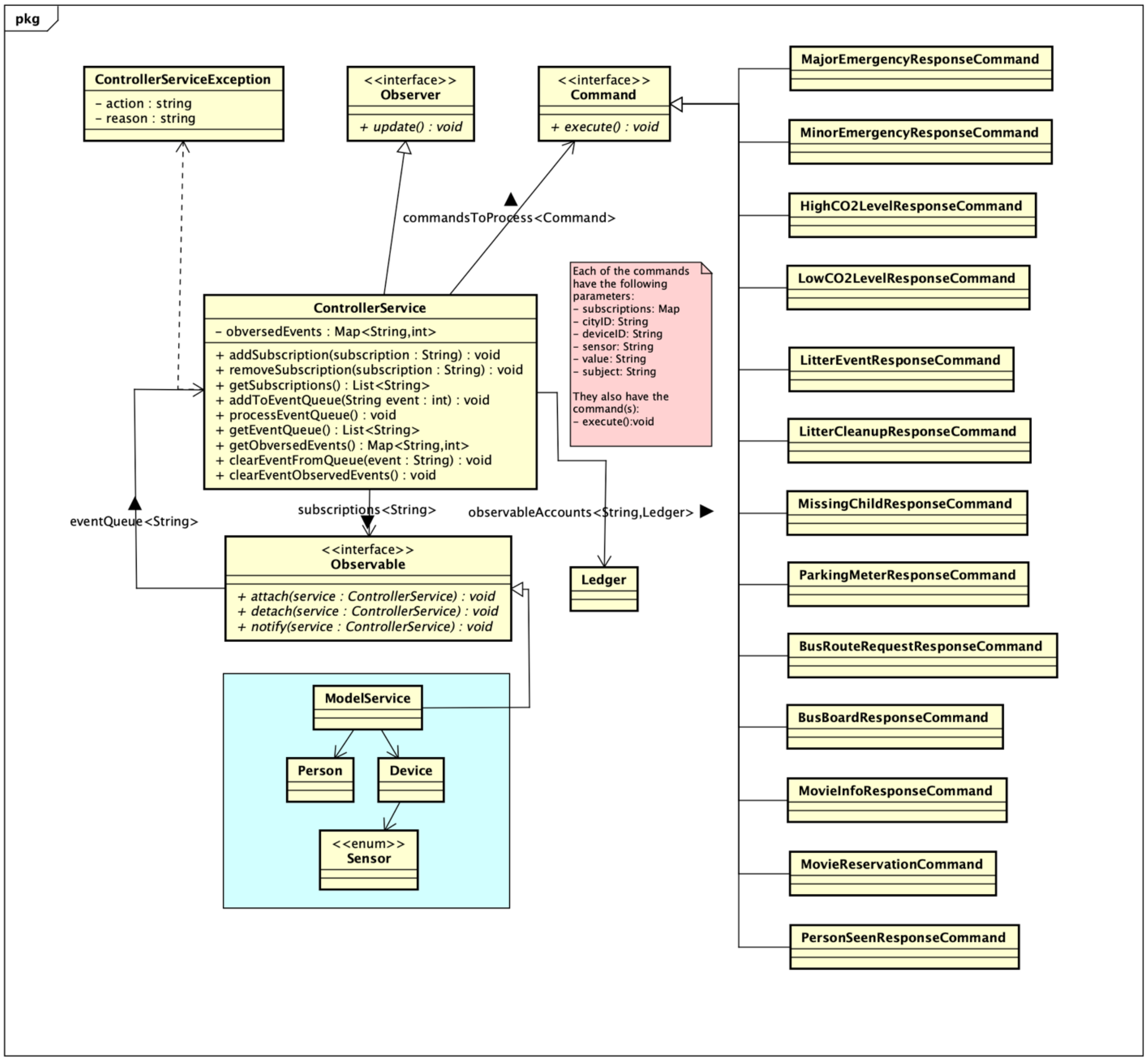
Devices are collections of individual sensors that can both communicate with the residents/visitors of the city, but also allow the controller to manage the city as a whole. It can subscribe to the controller and assist in managing their respective cities.

Implementation

This section of the document will describe the implementation details for the Controller Service.

Class Diagram

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


pkg
ControllerServiceException
– action : string
– reason : string
<<interface>>
Observer
+ update() : void
<<interface>>
Command
+ execute() : void
commandsToProcess<Command>
Each of the commands have the following parameters:
– subscriptions: Map
– cityID: String
– deviceID: String
– sensor: String
– value: String
– subject: String
They also have the command(s):
– execute():void
ControllerService
– obversedEvents : Map<String,int>
+ addSubscription(subscription : String) : void
+ removeSubscription(subscription : String) : void
+ getSubscriptions() : List<String>
+ addToEventQueue(String event : int) : void
+ processEventQueue() : void
+ getEventQueue() : List<String>
+ getObversedEvents() : Map<String,int>
+ clearEventFromQueue(event : String) : void
+ clearEventObservedEvents() : void
eventQueue<String>
subscriptions<String>
observableAccounts<String,Ledger>
<<interface>>
Observable
+ attach(service : ControllerService) : void
+ detach(service : ControllerService) : void
+ notify(service : ControllerService) : void
Ledger
ModelService
Person
Device
<<enum>>
Sensor
MajorEmergencyResponseCommand
MinorEmergencyResponseCommand
HighCO2LevelResponseCommand
LowCO2LevelResponseCommand
LitterEventResponseCommand
LitterCleanupResponseCommand
MissingChildResponseCommand
ParkingMeterResponseCommand
BusRouteRequestResponseCommand
BusBoardResponseCommand
MovieInfoResponseCommand
MovieReservationCommand
PersonSeenResponseCommand

Class Dictionary

This section specifies the class dictionary for the class.

Controller Class

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.

Methods

Method Name	Signature	Description
addSubscription	addSubscription(subscription:String):void	
removeSubscription	removeSubscription(subscription:String):void	
addToEventQueue	addToEventQueue(String event:int):void	
processEventQueue	processEventQueue():void	
clearEventQueue	clearEventQueue():void	
addToObservedEvents	addToObservedEvents(Map<String,String>):void	
processObservedEvents	processObservedEvents():void	
clearObservedEvents	clearObservedEvents():void	

Properties

Property Name	Signature	Description
observedEvents	observedEvents: Map<String,Int>	This represents the collected processed events and their occurances

Associations

Association Name	Signature	Description
subscriptions	subscriptions: List<Observable>	This is a collection of services that the Controller service is listening to
eventQueue	eventQueue: List<String>	This is a collection of event signatures sent to the Controller service
observableAccounts	observableAccounts: Map<String, Ledger>	This is a set of accounts associated with the observable elements and sub-elements

ControllerException Class

The ControllerException class captures any processing error originating within the Controller Service. This does not include ModelExceptions or LedgerExceptions.

Properties

Property Name	Signature	Description
action	action:String	
reason	reason:String	

Observer Interface

The Observer interface allows the Controller Service to manually update itself based on a series of events.

Methods

Method Name	Signature	Description
update	update():void	

Observable Interface

The Observable interface all objects to attach to an Observer service such that it can freely send updates to it without explicitly calling the service.

Methods

Method Name	Signature	Description
attach	attach(service: Controller)	
detach	detach(service: Controller)	
notify	notify(service: Controller)	

Command Interface

The command interface allows class commands to be executed without having to know what is truly being executed.

Methods

Method Name	Signature	Description
execute	execute():void	Allow any of its dependencies to freely execute a function on instantiation

MajorEmergencyResponseCommand Class

The MajorEmergencyResponseCommand handles large scale emergencies such as fires, floods, sever weather, and earthquakes, notifies the entire city, and deploys robots to help.

Methods

Method Name	Signature	Description
execute	execute():void	Given an emergency flag (i.e. fire, flood, earthquake, or severe weather) and their location, announce to the

		city “There is a <emergency type> in <city>, please find shelter”. Assure that half the robots address the issue, and that the remaining robots help people find shelter.

MinorEmergencyResponseCommand Class

The MinorEmergencyResponseCommand handles small scale emergencies such as traffic accidents, notifies the specified device, and deploys robots to help.

Methods

Method Name	Signature	Description
execute	execute():void	Given an emergency flag (i.e. traffic accident) and their location, announce to the reporting device “Stay calm, help is on its way”. Assure that the 2 nearest robots address the issue.

HighCO2LevelResponseCommand Class

The HighCO2LevelResponseCommand handles events where the CO2 level in the city may be too high (i.e. over 1000).

Methods

Method Name	Signature	Description
execute	execute():void	If reported by more than 3 devices within a city, disable all cars in the city.

LowCO2LevelResponseCommand Class

The LowCO2LevelResponseCommand handles events where the CO2 level in the city may be approaching a good value (i.e. under 1000).

Methods

Method Name	Signature	Description

execute	execute():void	If reported by more than 3 devices within a city, Enable all cars in the city.

LitterEventResponseCommand Class

The LitterEventResponseCommand handles events regarding people within the city littering.

Methods

Method Name	Signature	Description
execute	execute():void	If a person performs an action such as throwing garbage on the ground, notify the device to remind them not to litter, and send a robot to clean it up. Additionally, if possible, charge that person 50 units for littering.

LitterCleanupResponseCommand Class

The LitterCleanupResponseCommand handles events regarding garbage or potential garbage found within the city.

Methods

Method Name	Signature	Description
execute	execute():void	If the device hears the sound of broken glass at a location, send a robot to clean it up.

MissingChildResponseCommand Class

The MissingChildResponseCommand handles events where missing children or people go missing and need to be found.

Methods

Method Name	Signature	Description
execute	execute():void	In response to a request seeking someone, locate them within the city, notify the asking device,

		and send a robot to retrieve them.

PersonSeenResponseCommand Class

The PersonSeenResponseCommand handles events where a person is seen at a new location.

Methods

Method Name	Signature	Description
execute	execute():void	Update the person location to that of the viewing device.

ParkingMeterResponseCommand Class

The ParkingMeterResponseCommand handles events where a vehicle has been parked within a parking space for a specified amount of time.

Methods

Method Name	Signature	Description
execute	execute():void	In response to a vehicle being parked for 1 hour, charge the vehicle account the specified fee.

BusRouteRequestResponseCommand Class

The BusRouteRequestResponseCommand handles requests for bus routes.

Methods

Method Name	Signature	Description
execute	execute():void	In response to a person asking if the bus goes to particular location, respond “Yes, this bus goes to…”

BusBoardResponseCommand Class

The BusBoardResponseCommand handles transactions involved when someone boards a bus.

Methods

Method Name	Signature	Description

execute	execute():void	As a person boards the bus, if possible, charge them for the specific fee.

MovieInfoResponseCommand Class

The MovieInfoResponseCommand handles requests where someone asks about movie schedules.

Methods

Method Name	Signature	Description
execute	execute():void	If a person ask, "what movies are showing tonight?", respond with "Casablanca is showing at 9 pm", and set the display to: "https://en.wikipedia.org/wiki/Casablanca_(film)#/media/File:CasablancaPoster-Gold.jpg"

MovieReservationCommand Class

The MovieReservationCommand handles transactions related to movie reservations.

Methods

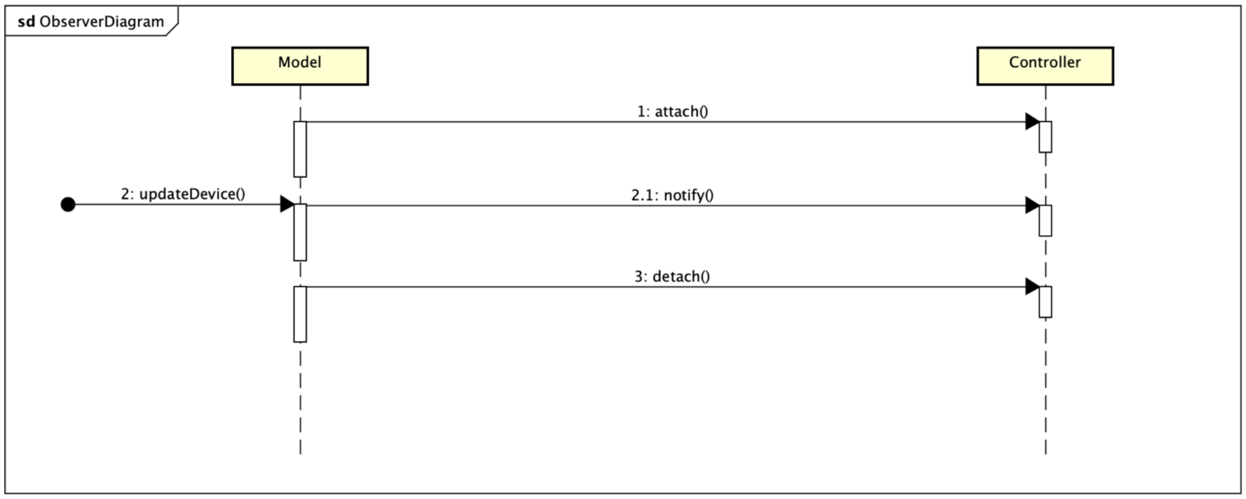
Method Name	Signature	Description
execute	execute():void	If a person says "reserve 2 seats for the 9 pm showing of Casablanca", if possible, charge them 10 units and respond "your seats are reserved; please arrive a few minutes early."

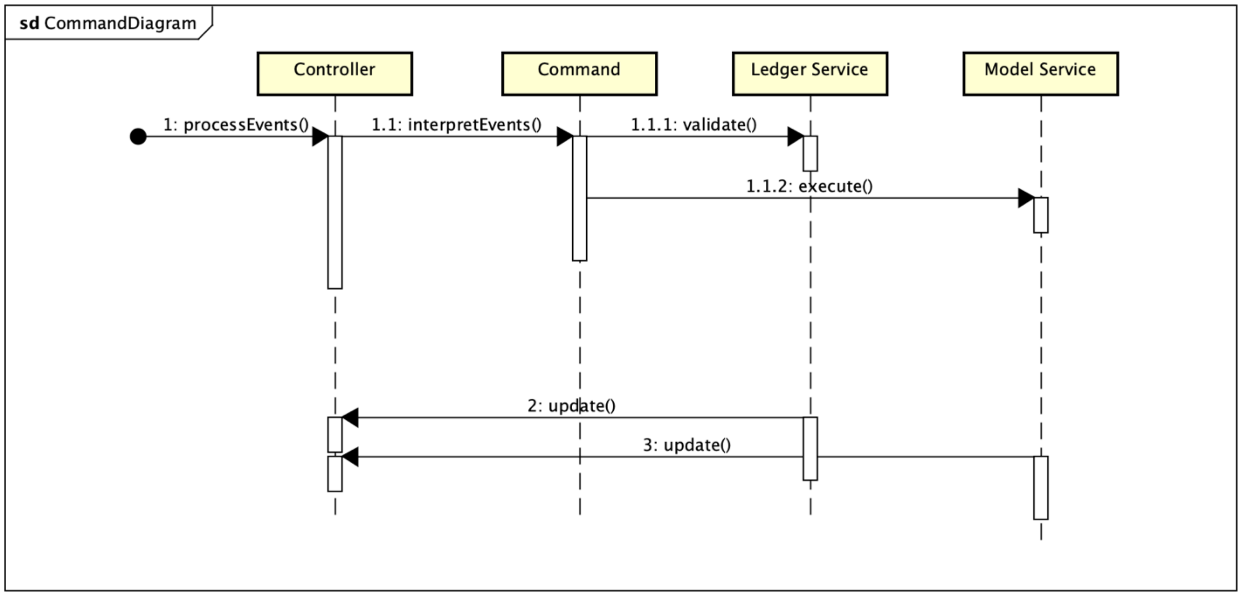
Implementation Details

Below, we have Sequence diagrams that describe the general flow from the observer viewpoint and the commander viewpoint.

From the observer view point, we see that upon establishment a controller attachment, the model's cities and devices should not be able to freely send sensor events to the Controller service. Additionally, it displays that upon detachment, this feature is no longer possible or trackable.

From the commander view point, we see that upon receiving a batch of events from the observable services, they need to be processed. This processing ultimately counts the number of related events, and upon determining this, based on the specifications, will propagate a command to be executed in response. This will update the observed objects to assure that their issue has been addressed.





Exception Handling

In addition to an exception brought on by the Model and Ledger service, the primary exception considered in this service are the Controller Exception. The controller exception is thrown whenever any process in restructuring items in the event flow fail, such as when the Observer and Command interface functions fail due to missing or poorly structured input.

Testing

NOTE: For the sake of testing, I have given every person and device 1000 units such that they are able to perform the financially impactful activities within the city.

NOTE: For the sake of testing, I have bound the update and notify function to faster testing. The two do not need to be bound.

Below, we have a list of testing strategies required and provided by this service:

- Functional Test
  - Given a pre-defined list off all commands and assure that they function properly and are handled appropriately.
- Performance
  - Assure that there are no hanging processes within the service and that no process runs any longer than is necessary.
- Regression
  - Assure that the inclusion of new functionalities and features do not ultimately bring about errors that did not exist before.
- Exception Handling
  - Capture the exceptions defined above, and return this in a human friendly, but not overly revealing, context.

Risks

The risks that I identified in the process of this design, were:

- The is no well-defined method of how we want to organize accounts across the individual cities or the entire model service.
- The is no way to determine what certain elements are not present within the Observable objects (a model without any cities, or a city without people, devices, etc).
- There is no specified time interval for which notifications and updates should happen.