Smart Transportation System Design Document

Date: 11/17/2020
Author: Christopher Jones
Reviewer(s):

- Ethan Anthony
- Hidai Bar-Mor

Table of Contents

Each of the following sections and subsections can be following on the following pages below:

SMART TRANSPORTATION SERVICE

Introduction

This document defines the overall design for the Smart Transportation System.

Overview

In our previous assignments, we designed services for modeling/simulation (*Model Service*), financial management (*Ledger Service*), security (*Authentication Service*), and lastly, monitoring (*Controller Service*). However, these services ultimately oversee a static system unless provided near live data and updates. This is where the smart transportation system comes in.

This system is made up of a series of modules, discussed later in this document that, together, simulate a real-world ride request system from account setup all the way to trip completion. These modules will be using the previous services designed to construct a functional “smart” transportation system. The leveled component diagram below displays the interactions between each of the services.

Requirements

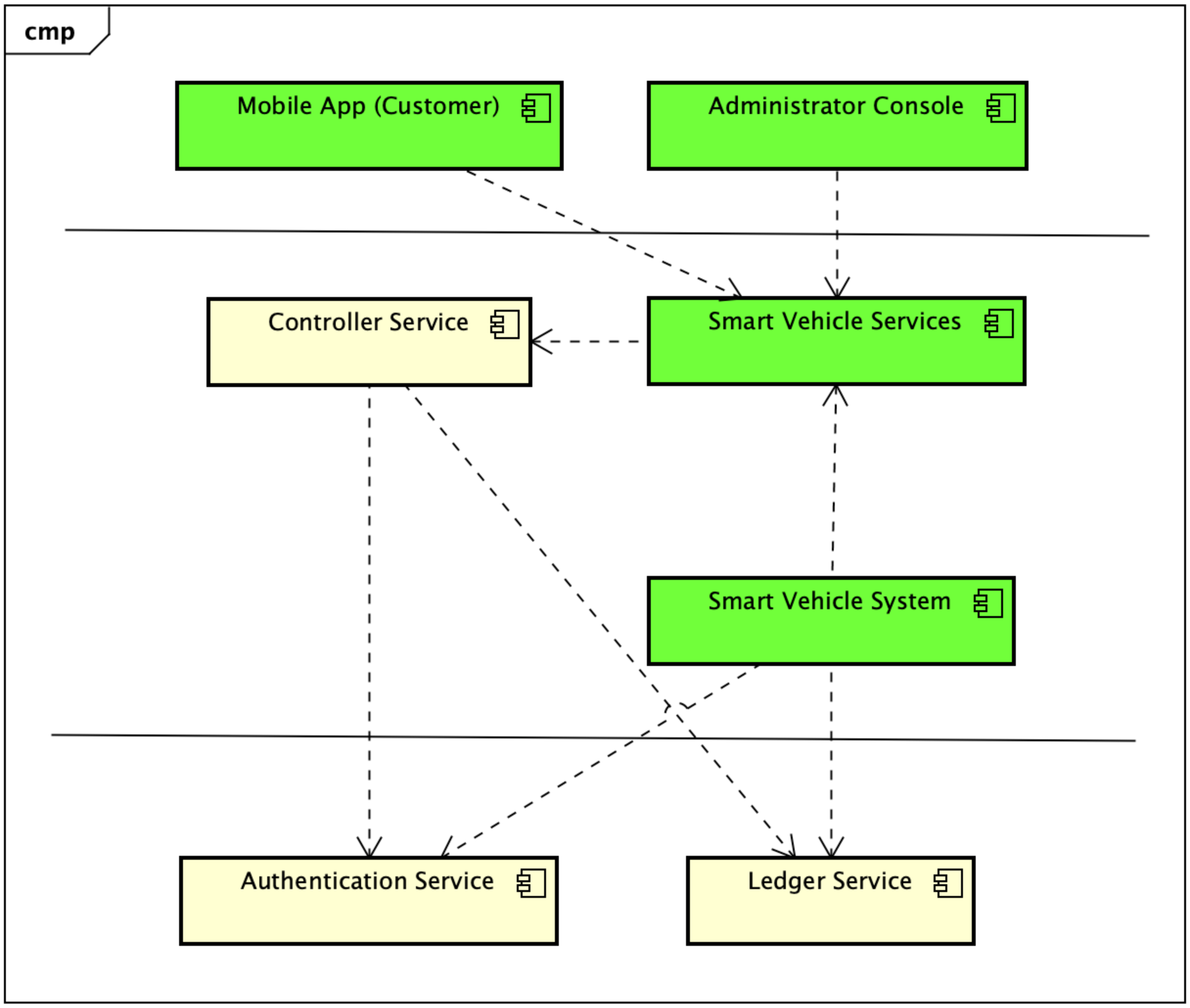
This section provides a brief summary of the requirements for the Smart Transportation System.

The Smart Transportation System is responsible for assuring that users can create ride requests from individual vehicles, and if they are available to do so, the vehicles should process and complete those requests. This system is comprised of 4 modules that call interact with the services we’ve provided in previous assignments.

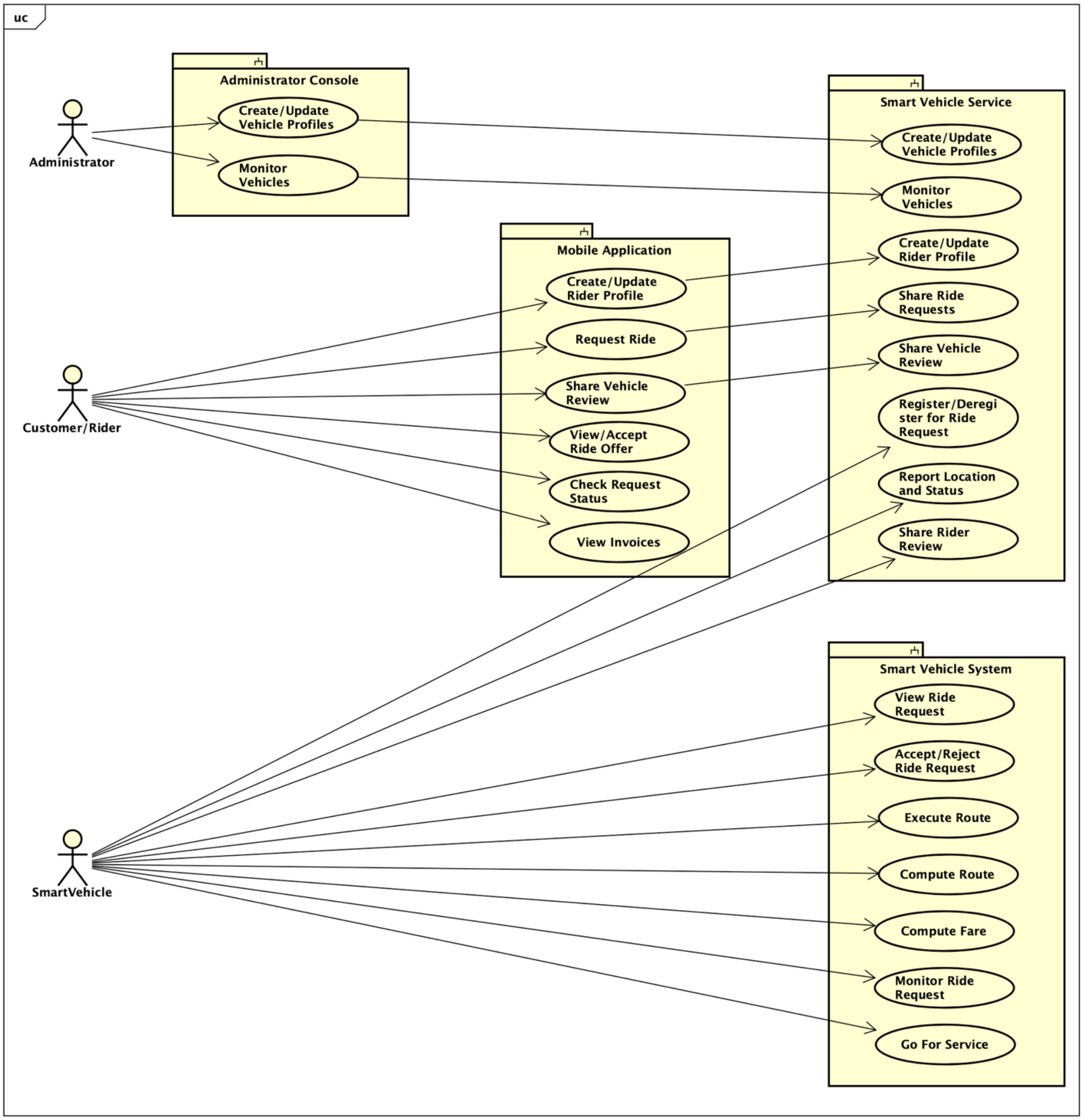
The Smart Transportation System utilizes the following 4 modules, namely:

- Administration Interface
- Customer Interface
- Smart Vehicle Service
- Smart Vehicle System

The following leveled module component diagram describes their interaction with each other and the other services.


cmp
Mobile App (Customer)
Administrator Console
Controller Service
Smart Vehicle Services
Smart Vehicle System
Authentication Service
Ledger Service

Use Cases



In this system, there are only 3 types of users:

- Administrator
- Customer/Rider
- Smart Vehicle

Administrators define and managed the suite of vehicles within the city. They not only provide those vehicles to customers/riders to request rides from, but they also perform a series of tasks to monitor them both as well. Administrators keep an eye on vehicle's service schedules, locations, history, routes, and ratings, as well as customer's history, requests, location, and ratings. Their primary role is to ensure that the smart transportation is operating efficiently and effectively.

Customers and Riders are the primary consumers of the services that the vehicles provide. They manage their own profiles and request rides at their own leisure. After registering, they can freely request rides (provided that they are financially able), accept ride offers, view and review individual vehicles, and manage their invoices.

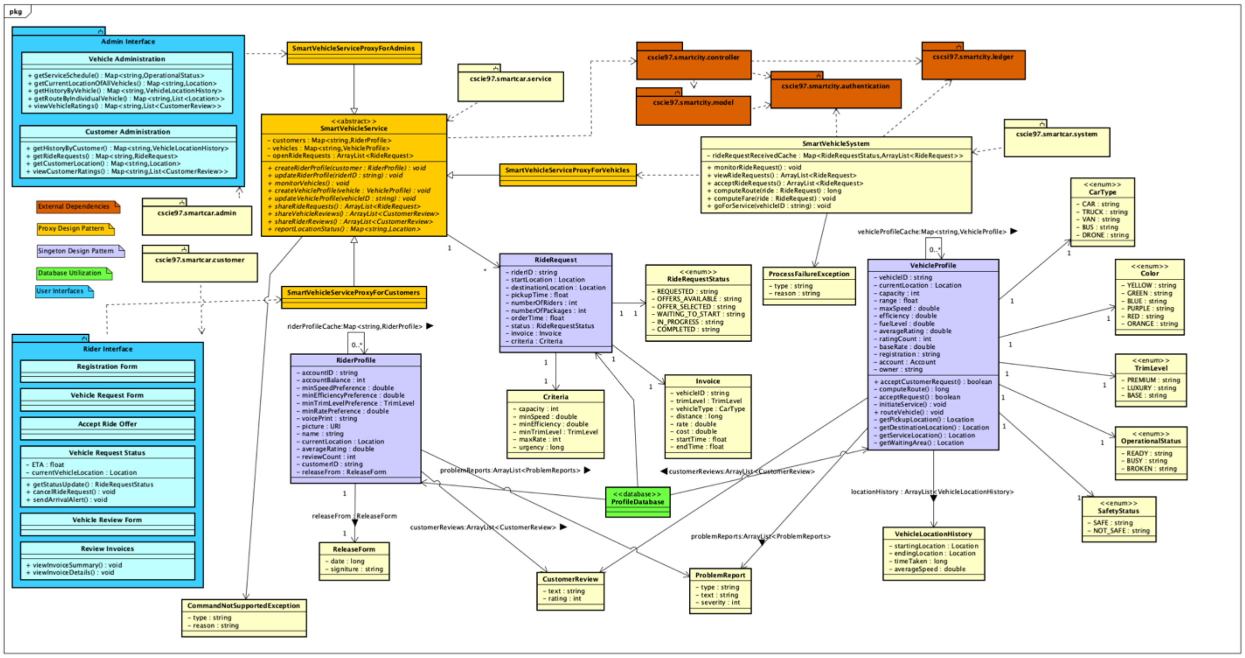
Smart Vehicles are the primary service provides in this system. They take in requests, from Customers/Riders, offer them rides if they are able to perform it, and transport them to the location specified. Vehicles have varying degrees of operational and safety statuses, and can get services to resolve any issues that arise with them.

Implementation

The section of the document displays a high-level overview of the details of the Smart Transportation System. These modules will be discussed more in-depth later in this document.

Class Diagram

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



pkg
Admin Interface
Vehicle Administration
+ getServiceSchedule() : Map<string,OperationalStatus>
+ getCurrentLocationOfAllVehicles() : Map<string,Location>
+ getHistoryByVehicle() : Map<string,VehicleLocationHistory>
+ getRouteByIndividualVehicle() : Map<string,List<Location>>
+ viewVehicleRatings() : Map<string,List<CustomerReview>>
Customer Administration
+ getHistoryByCustomer() : Map<string,VehicleLocationHistory>
+ getRideRequests() : Map<string,RideRequest>
+ getCustomerLocation() : Map<string,Location>
+ viewCustomerRatings() : Map<string,List<CustomerReview>>
External Dependencies
Proxy Design Pattern
Singeton Design Pattern
Database Utilization
User Interfaces
cscie97.smartcar.admin
cscie97.smartcar.customer
SmartVehicleServiceProxyForAdmins
cscie97.smartcar.service
<<abstract>>
SmartVehicleService
- customers : Map<string,RiderProfile>
- vehicles : Map<string,VehicleProfile>
- openRideRequests : ArrayList<RideRequest>
+ createRiderProfile(customer : RiderProfile) : void
+ updateRiderProfile(riderID : string) : void
+ monitorVehicles() : void
+ createVehicleProfile(vehicle : VehicleProfile) : void
+ updateVehicleProfile(vehicleID : string) : void
+ shareRideRequests() : ArrayList<RideRequest>
+ shareVehicleReviews() : ArrayList<CustomerReview>
+ shareRiderReviews() : ArrayList<CustomerReview>
+ reportLocationStatus() : Map<string,Location>
cscie97.smartcity.controller
cscie97.smartcity.model
cscie97.smartcity.authentication
cscsi97.smartcity.ledger
cscie97.smartcar.system
SmartVehicleServiceProxyForVehicles
SmartVehicleSystem
- rideRequestReceivedCache : Map<RideRequestStatus,ArrayList<RideRequest>>
+ monitorRideRequest() : void
+ viewRideRequests() : ArrayList<RideRequest>
+ acceptRideRequests() : ArrayList<RideRequest>
+ computeRoute(ride : RideRequest) : long
+ computeFare(ride : RideRequest) : void
+ goForService(vehicleID : string) : void
<<enum>>
CarType
- CAR : string
- TRUCK : string
- VAN : string
- BUS : string
- DRONE : string
vehicleProfileCache:Map<string,VehicleProfile>
0..*
VehicleProfile
- vehicleID : string
- currentLocation : Location
- capacity : int
- range : float
- maxSpeed : double
- efficiency : double
- fuelLevel : double
- averageRating : double
- ratingCount : int
- baseRate : double
- registration : string
- account : Account
- owner : string
+ acceptCustomerRequest() : boolean
- computeRoute() : long
- acceptRequest() : boolean
- initiateService() : void
+ routeVehicle() : void
- getPickupLocation() : Location
- getDestinationLocation() : Location
- getServiceLocation() : Location
- getWaitingArea() : Location
<<enum>>
Color
- YELLOW : string
- GREEN : string
- BLUE : string
- PURPLE : string
- RED : string
- ORANGE : string
<<enum>>
TrimLevel
- PREMIUM : string
- LUXURY : string
- BASE : string
<<enum>>
OperationalStatus
- READY : string
- BUSY : string
- BROKEN : string
<<enum>>
SafetyStatus
- SAFE : string
- NOT_SAFE : string
locationHistory : ArrayList<VehicleLocationHistory>
VehicleLocationHistory
- startingLocation : Location
- endingLocation : Location
- timeTaken : long
- averageSpeed : double
RideRequest
- riderID : string
- startLocation : Location
- destinationLocation : Location
- pickupTime : float
- numberOfRiders : int
- numberOfPackages : int
- orderTime : float
- status : RideRequestStatus
- invoice : Invoice
- criteria : Criteria
<<enum>>
RideRequestStatus
- REQUESTED : string
- OFFERS_AVAILABLE : string
- OFFER_SELECTED : string
- WAITING_TO_START : string
- IN_PROGRESS : string
- COMPLETED : string
ProcessFailureException
- type : string
- reason : string
Criteria
- capacity : int
- minSpeed : double
- minEfficiency : double
- minTrimLevel : TrimLevel
- maxRate : int
- urgency : long
Invoice
- vehicleID : string
- trimLevel : TrimLevel
- vehicleType : CarType
- distance : long
- rate : double
- cost : double
- startTime : float
- endTime : float
SmartVehicleServiceProxyForCustomers
riderProfileCache:Map<string,RiderProfile>
0..*
RiderProfile
- accountID : string
- accountBalance : int
- minSpeedPreference : double
- minEfficiencyPreference : double
- minTrimLevelPreference : TrimLevel
- minRatePreference : double
- voicePrint : string
- picture : URI
- name : string
- currentLocation : Location
- averageRating : double
- reviewCount : int
- customerID : string
- releaseFrom : ReleaseForm
problemReports:ArrayList<ProblemReports>
customerReviews:ArrayList<CustomerReview>
<<database>>
ProfileDatabase
releaseFrom : ReleaseForm
ReleaseForm
- date : long
- signiture : string
customerReviews:ArrayList<CustomerReview>
problemReports:ArrayList<ProblemReports>
CustomerReview
- text : string
- rating : int
ProblemReport
- type : string
- text : string
- severity : int
Rider Interface
Registration Form
Vehicle Request Form
Accept Ride Offer
Vehicle Request Status
- ETA : float
- currentVehicleLocation : Location
+ getStatusUpdate() : RideRequestStatus
+ cancelRideRequest() : void
+ sendArrivalAlert() : void
Vehicle Review Form
Review Invoices
+ viewInvoiceSummary() : void
+ viewInvoiceDetails() : void
CommandNotSupportedException
- type : string
- reason : string
1
*

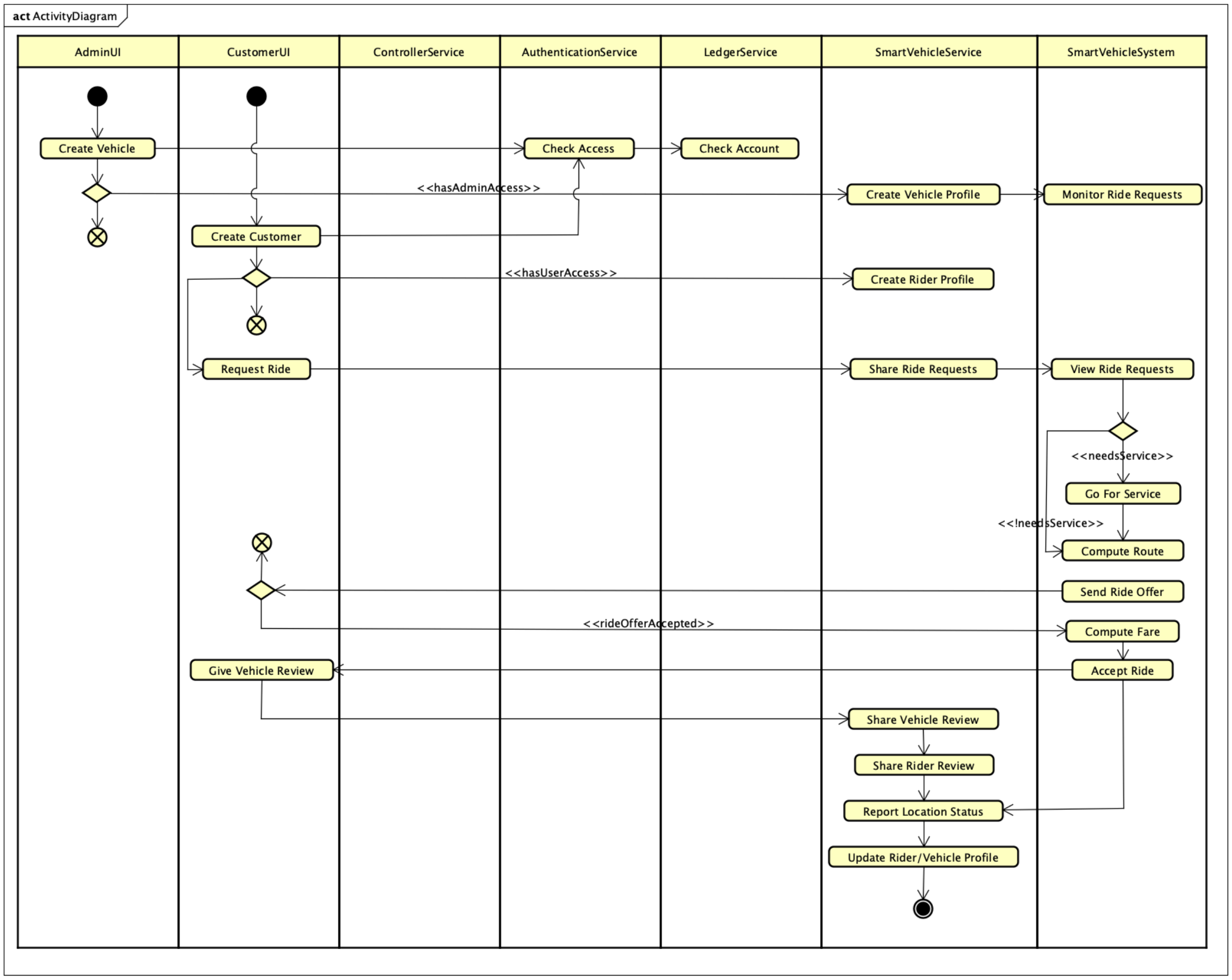
Class Dictionary

The class dictionary will be broken down by module in the upcoming sections.

Implementation Details

While relevant sequence diagrams will be provided later in this document, the overall scope is to allow various subcomponents to work together and effectively divide and conquer the work necessary to reach a similar goal, that is, getting an individual from one location to another.

Below, we utilize the following Activity diagram to describe the overall flow of the system we are trying to create from the perspectives of both an administrator and a rider/customer from start to finish.



As you can see, they expectation is that the process of any given user who does use the service, without error, should have their paths merged with that of an existing vehicle.

In this process, design patterns such as the Command, Proxy the Singleton method, and on occasion, a database for persistence, to help manage the varying parts within the system. In the existence of such a data base, we'd store the following:

- Vehicle Profiles
- Rider Profiles
- Ride Requests
  - Open
  - Pending
  - Closed

Each item stored would essentially be a high access and high update element.

Exception Handling

While other exceptions exist as a result of the other services, the primary exceptions handled with respect to the transportation system are:

- CommandNotSupportedException
- ProcessFailureException

The **CommandNotSupportedException** accounts for any requested action performed by an individual with invalid access rights (*i.e., riders who attempt to create a vehicle profile*).

The **ProcessFailureException** accounts for any failure within the system itself. Many examples can trigger this exception (*i.e., no vehicles existing for a specific set of specifications, no vehicles existing at all, failed payment process, etc*.).

Exceptions captured from external services should be compartmentalized and provided through the ProcessFailureException.

Testing

Testing this application should be tested in a similar manner as the previous projects, except an E2E testing framework may need to be included to test the elements of the UI to validate the full process. Additionally, we will need tests that perform the following:

- Functional Ability
  - Given a pre-defined list of actions, assure they reach a pre-defined list of results
- Performance

    - Since this is to be a fully automated system, we must assure that no hanging processes exist that could block another process
- Regression
    - We need to assure that the inclusion of new functionalities and features do not bring about errors that did not exist before
- Exception Handling
    - We need to assure that all related exceptions are caught in the events that we expect to trigger them

Risks

Some potential risks identified within this system is:

- There is no component preventing the administrator user from accessing the customer account information.
- Cross-city transportation may introduce issues if not accounted for.
- The current system does not address vehicles with consistent low ratings and poor reviews.
- The database and caching systems have no methods of notifying the end user when it is full or down.

SMART VEHICLE SERVICE

Introduction

This section defines the design for the Smart Vehicle Service.

Overview

The Smart Vehicle Service is the main entry point for both administrators and customers to provide and manage the primary entities of the system, namely profiles, requests, and updates. Each of these elements can be persisted with the use of an external database as well to improve performance. Overall, this system cannot be considered a functionable system without actionable objects and processes. This service provides each of those distinct underlying elements to be acted upon later.

Requirements

This section provides a brief summary of the requirements for the Smart Vehicle Service.

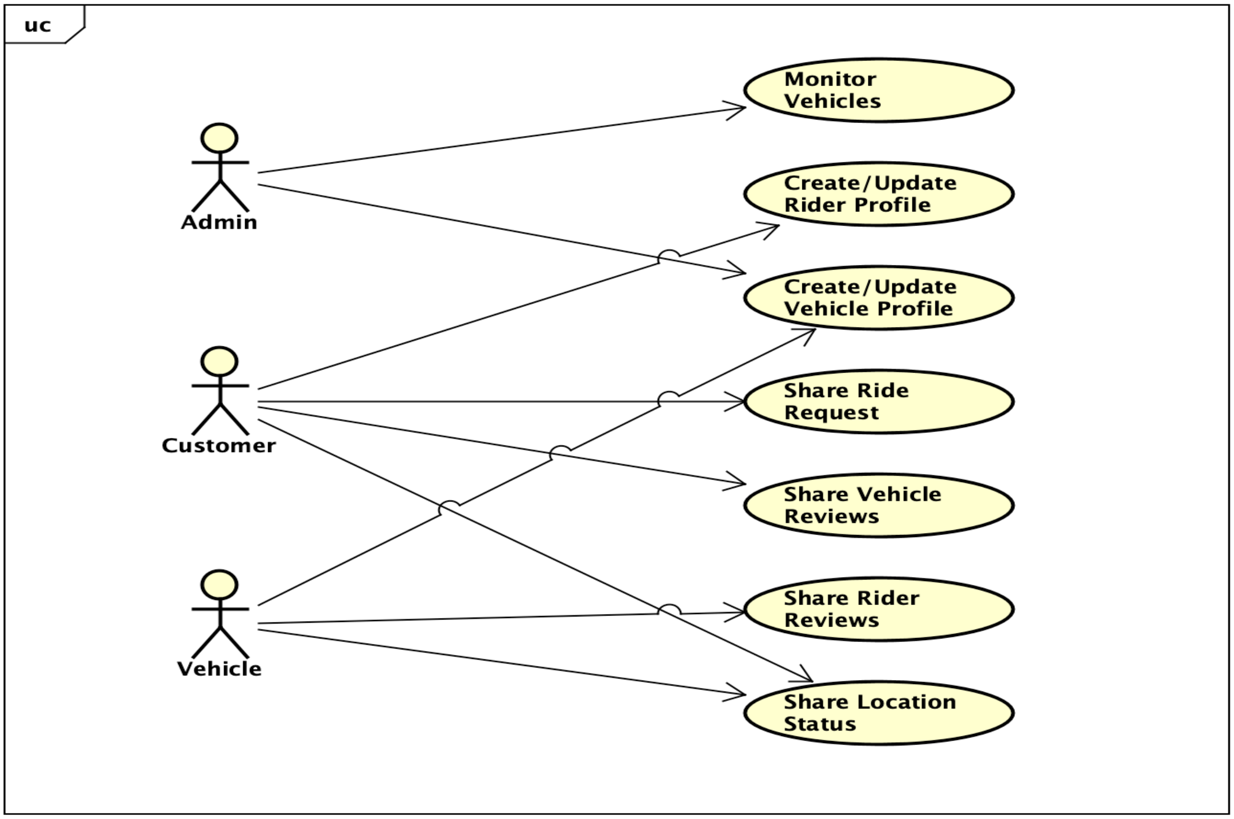
The Smart Vehicle Service is responsible for assuring that vehicles exist and are operational, that customers are able to register and request services, and that reviews can be provided my each to assure the overall success of the system.

The Smart Vehicle Service utilizes the following entities:

- Vehicle Profiles
    - Identification
        - Vehicle ID, Owner, Account, Registration
    - Specifications
        - Car Type, Color, Trim Level, Capacity, Range, Maximum Speed, Efficiency, Base Rate
    - Current Status
        - Current Location, Fuel Level, Operational Status, Safety Status
    - History
        - Location, Problem Reports, Customer Reviews, Average Rating, Rating Count
- Rider Profiles
    - Identification
        - Account ID, Customer ID, Voice Print, Picture, Name
    - Current Status
        - Account Balance, Location, Registration Form

  - Preferences
    - Minimum Speed, Minimum Efficiency, Minimum Trim Level, Minimum Rate Preference
  - History
    - Average Rating, Review Count
- Ride Requests
  - Identification
    - Rider ID
  - Specifications
    - Location
      - Start, Destination
    - Times
      - Pick Up Time, Order Time
    - Number of Riders/Packages
    - Criteria
  - Actionable Elements
    - Invoice
  - Delivery Cache

Use Cases



In this system, there are only 3 types of users:

- Administrator
- Customer
- Vehicle

Administrators define and managed the suite of vehicles within the city. They provide those vehicles to customers to request rides from and update their profiles respectively.

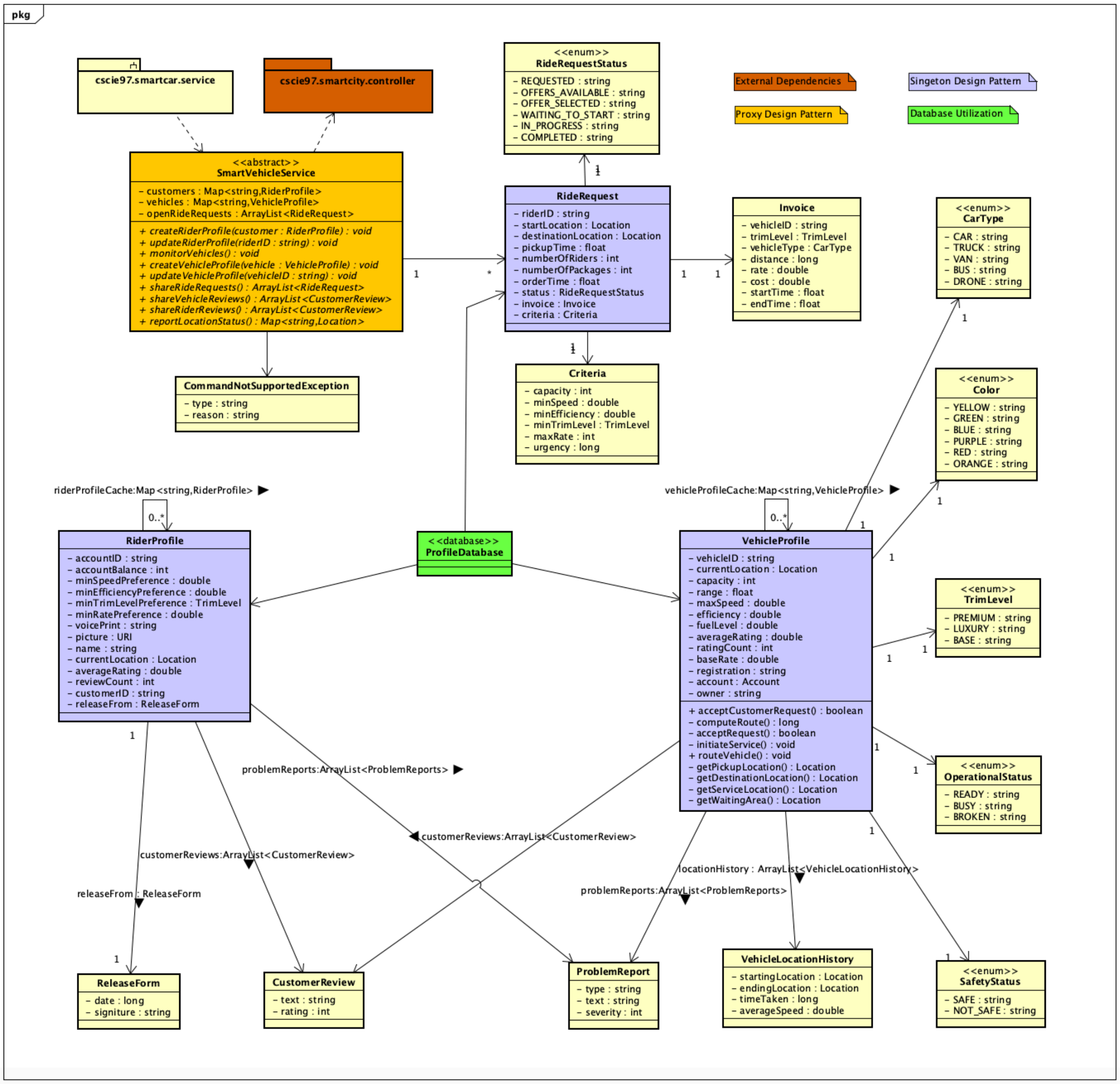
Customers are the primary consumers of the services that the vehicles provide. They manage their own profiles and request rides at their own leisure. They keep their own profiles up to date upon completion of a ride from a vehicle.

Smart Vehicles are the primary service provides in this system. They provide the capability to update their own profiles, share reviews, and notify end users of their locations.

Implementation

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

Class Diagram



Class Dictionary

This section specifies the classes for the given package.

SmartVehicleService Class

This class is supposed to operate as the main entry point, following the console and interface, into the Smart Transportation System. Here, all actionable elements are created and managed, and either persisted through in-memory caching, or through an external database.

Methods

Method Name	Signature	Description
createRiderProfile	createRiderProfile(customer:RiderProfile):void	Creates a new Rider Profile
updateRiderProfile	updateRiderProfile(riderID:string):void	Updates an existing Rider Profile, and throws an Exception otherwise
monitorVehicles	monitorVehicles():void	Prints out list of Vehicles in the system
createVehicleProfile	createVehicleProfile(vehicleID:string):void	Creates a new Vehicle Profile
updateVehicleProfile	updateVehicleProfile(vehicleID:string):void	Updates an existing Vehicle Profile, and throws an Exception otherwise
shareRideRequests	shareRideRequests():List<RideRequest>	Send out new ride request list
shareVehicleReviews	shareVehicleReviews():List<CustomerReview>	Send out vehicle reviews list
shareRiderReviews	shareRiderReviews():List<CustomerReview>	Send out rider reviews list
reportLocationStatus	reportLocationStatus():Map<string,Location>	Update the location of Rider or Vehicle

Properties

Method Name	Signature	Description
customers	Customers:Map<string,RiderProfile>	In-memory cache of Rider Profiles within the system
vehicles	Vehicles: Map<string,VehicleProfile>	In-memory cache of Vehicle Profiles within the system

Associations

Method Name	**Signature**	**Description**
openRideRequests	openRideRequests:List<RideRequest>	In-memory cache of new Ride Requests within the system

RiderProfile Class

This class identifies a customer/rider within the system. This class services to identify minimum qualifications for all interacting services.

Properties

Method Name	**Signature**	**Description**
accountID	accountID:string	The account ID of the Rider
accountBalance	accountBalance:int	The account balance of the Rider
minSpeedPreference	minSpeedPreference : double	Minimum speed of requested vehicles
minEfficiencyPreference	minEfficiencyPreference : double	Minimum efficiency of requested vehicles
minTrimLevelPreference	minTrimLevelPreference : TrimLevel	Minimum trim level of requested vehicles
minRatePreference	minRatePreference : double	Minimum rate of requested vehicles
voicePrint	voicePrint : string	The voice print of the Rider
picture	picture : URI	The profile picture of the Rider
name	name : string	The name of the Rider
currentLocation	currentLocation : Location	The current location of the Rider
averageRating	averageRating : double	The average rating given by the Rider
reviewCount	reviewCount : int	The reviews given by the Rider
customerID	customerID : string	The customer ID of the Rider
releaseFrom	releaseFrom : ReleaseForm	The form the user uses to register with a vehicle

Associations

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

riderProfileCache	riderProfileCache:Map<string,RiderProfile>	In-memory cache of Rider profiles
customerReviews	customerReviews:ArrayList<CustomerReview>	In-memory cache of Customer Reviews
problemReports	problemReports:ArrayList<ProblemReports>	In-memory cache of Problem Reports

VehicleProfile Class

This class identifies a vehicle within the system. They can pair with customers/rides, for whom they fulfill the minimum requirements for, to delivery either them, or packages, around the city.

Methods

Method Name	**Signature**	**Description**
acceptCustomerRequest	acceptCustomerRequest() : boolean	Returns true or false based on whether the vehicle isn’t already processing a ride
computeRoute	computeRoute() : List<Location>	Organize list List of Locations as coordinates as stops
acceptRequest	acceptRequest() : boolean	Returns true or false based on whether the customer has a positive balance, the vehicle has enough fuel, and if all status is in good standing
initiateService	initiateService() : void	Convert all status values to good standing and reset fuel level to maximum value
routeVehicle	routeVehicle() : void	Route Vehicle to the next stop if it has one
getPickupLocation	getPickupLocation() : Location	Get location and check if it is in route or not. If not, if the ride request is accepted, add it to the route
getDestinationLocation	getDestinationLocation() : Location	Get location and check if it is in route or not. If not, if the ride request is accepted, add it to the route
getServiceLocation	getServiceLocation() : Location	Get location of nearest service stop

getWaitingArea	getWaitingArea() : Location	Get location of nearest waiting area

Properties

Method Name	Signature	Description
vehicleID	vehicleID : string	The ID for the Vehicle
currentLocation	currentLocation : Location	The current location of the vehicle
capacity	capacity : int	The maximum capacity of the vehicle
range	range : float	The maximum range of the vehicle
maxSpeed	maxSpeed : double	The maximum speed of the vehicle
efficiency	efficiency : double	The maximum efficiency of the vehicle
fuelLevel	fuelLevel : double	The maximum fuel level of the vehicle
averageRating	averageRating : double	The average rating given to the vehicle
ratingCount	ratingCount : int	The number of ratings given to the vehicle
baseRate	baseRate : double	The minimum rate a vehicle will charge
registration	registration : string	The signature of the most recent accepted rider
account	account : Account	The account of the vehicle
owner	owner : string	The name of the owner

Associations

Method Name	Signature	Description
customerReviews	customerReviews:ArrayList<CustomerReview>	In-memory cache of Customer Reviews
problemReports	problemReports:ArrayList<ProblemReports>	In-memory cache of Problem Reports

CommandNotSupportedException Class

This exception is to be throw whenever a function is called that should not be accessible by the caller.

Properties

Method Name	**Signature**	**Description**
type	type : string	Type of exception
reason	reason : string	Cause of the exception

Invoice Class

This class represents an invoice associated with a ride request.

Properties

Method Name	**Signature**	**Description**
vehicleID	vehicleID : string	The ID of the source vehicle
trimLevel	trimLevel : TrimLevel	The Trim level of the source vehicle
vehicleType	vehicleType : CarType	The Type of the source vehicle
distance	distance : long	The distance travelled
rate	rate : double	The rate of the trip
cost	cost : double	The cost of the trip
startTime	startTime : float	The trip start time
endTime	endTime : float	The trip end time

Criteria Class

This class identifies the minimum criteria to filter through vehicles to pair with the customer/rider.

Properties

Method Name	**Signature**	**Description**
capacity	capacity : int	Capacity request of the source ride request
minSpeed	minSpeed : double	Minimum speed requested from source request
minEfficiency	minEfficiency : double	Minimum efficiency requested from source request
minTrimLevel	minTrimLevel : TrimLevel	Minimum trim level requested from source request
maxRate	maxRate : int	Maximum rate requested from source request
urgency	urgency : long	Minimum urgency requested from source request

VehicleLocationHistory Class

This class represents the location history of vehicle.

Properties

Method Name	Signature	Description
startingLocation	startingLocation : Location	The location of vehicle at the start of a trip
endingLocation	endingLocation : Location	The location of vehicle at the end of a trip
timeTaken	timeTaken : long	The overall amount of time/stops taking to complete trip
averageSpeed	averageSpeed : double	The average speed requestive

ProblemReport Class

This class represents the reporting of an issue that would take the vehicle out of good standing, provoking to be flagged for service-needed thereafter.

Properties

Method Name	Signature	Description
type	type : string	The type of report
text	text : string	The text describing the problem
severity	severity : int	A number indicating the severity

CustomerReview Class

This class represents customer reviews left for any given vehicle. This determines how positive the overall experience was.

Properties

Method Name	Signature	Description
text	text : string	The text describing the trip
rating	rating : int	A number indicating the experience (0 being bad, 5 being great)

ReleaseForm Class

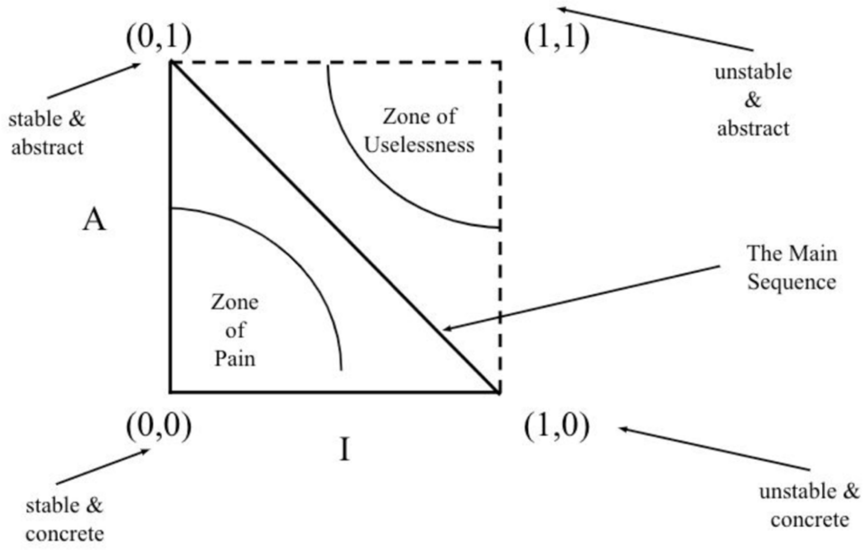
This class allows the rider and vehicle agree upon a specified ride under contract.

Properties

Method Name	Signature	Description
date	date : long	The current date
signature	signature : string	The signature of the Rider

Implementation Details

The implementation details are provided below.



Instability

Let C_a be the number of classes outside of a package that depend on one or more classes inside the package. These are called afferent classes

Let C_e be the number of classes that one or more of the classes within the given package depend on. These are called efferent classes:

$$\boldsymbol{I = \frac{C_e}{C_e + C_a}}$$

The instability of this class is computed by the following:

$$\boldsymbol{I = \frac{0}{0 + (C_{a_{service}} + C_{a_{admin}} + C_{a_{rider}})} = 0}$$

Abstraction

Let N_c be the number of classes and interfaces in the package

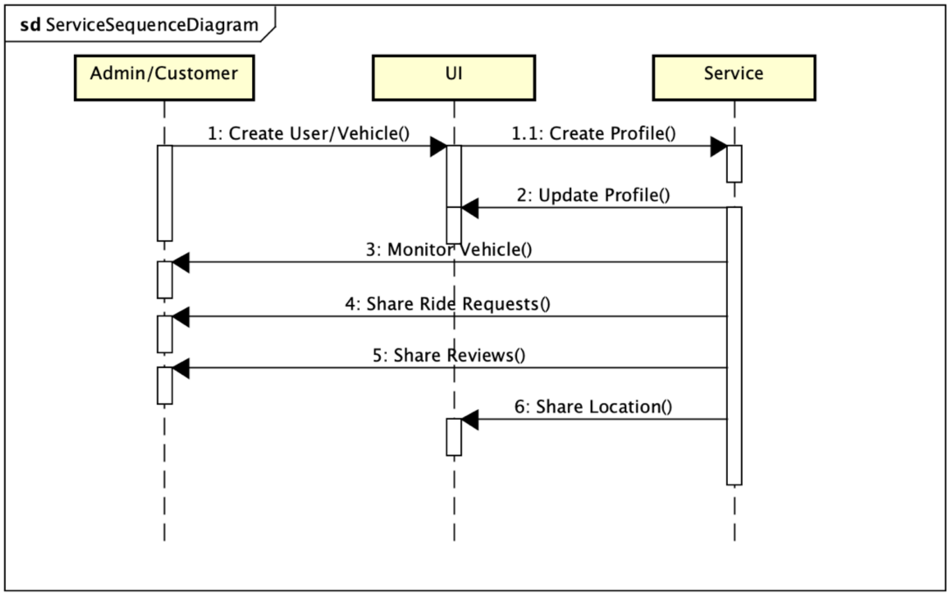
Let N_a be the number of the of non-instantiable classes and interfaces:

$$\boldsymbol{A = \frac{N_a}{N_c}}$$

The abstractness of this class is computed by the following:

$$\boldsymbol{A = \frac{(N_{a_{service}})}{N_{c_{service}} + N_{c_{profile}} + N_{c_{request}}} = \frac{1}{3}}$$

This would suggest that this package stable and, to a small degree, abstract. The sequence diagram below displays the sequence of events by the utilization of this package.



Exception Handling

While other exceptions exist as a result of the other services, the primary exceptions handled with respect to the this package are:

- CommandNotSupportedException
- ProcessFailureException

The **CommandNotSupportedException** accounts for any requested action performed by an individual with invalid access rights (*i.e., riders who attempt to create a vehicle profile*).

SMART VEHICLE SYSTEM

Introduction

This section defines the design for the Smart Vehicle System.

Overview

The Smart Vehicle System is the main manager of ride requests generated by the customers. It can view, monitor, and even accept/reject through checking account access, security credentials, and vehicle status. This system processes the ride requests and, on occasion, sends the vehicles for service in the event of any issue arising.

Requirements

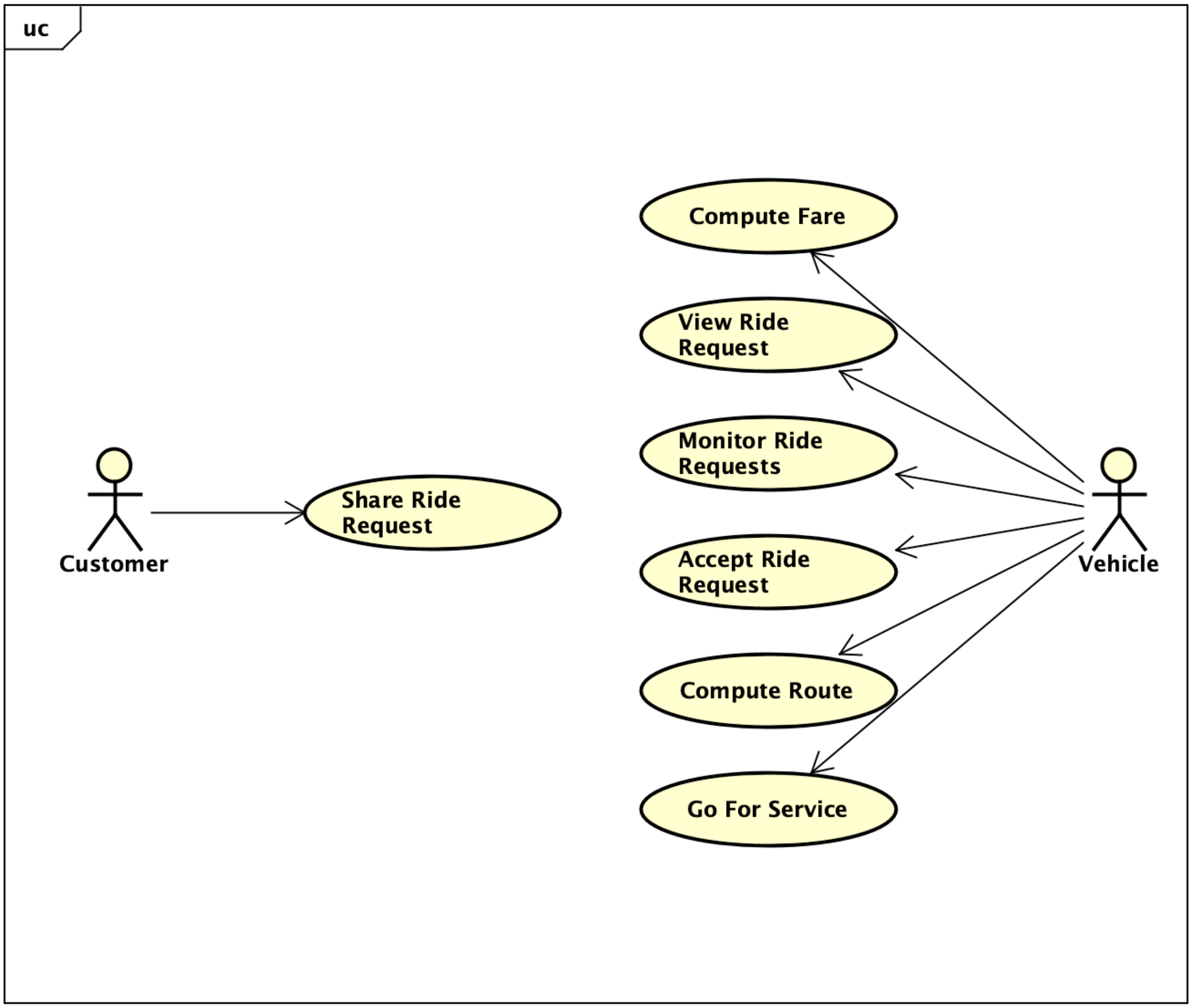
This section provides a brief summary of the requirements for the Smart Vehicle System.

The Smart Vehicle System is responsible for assuring that all ride requests, ride offerings, and service updates are processed properly.

The Smart Vehicle System utilizes the following entities:

- Ride Requests
  - Received Cache
- Vehicles
  - Associated Profiles

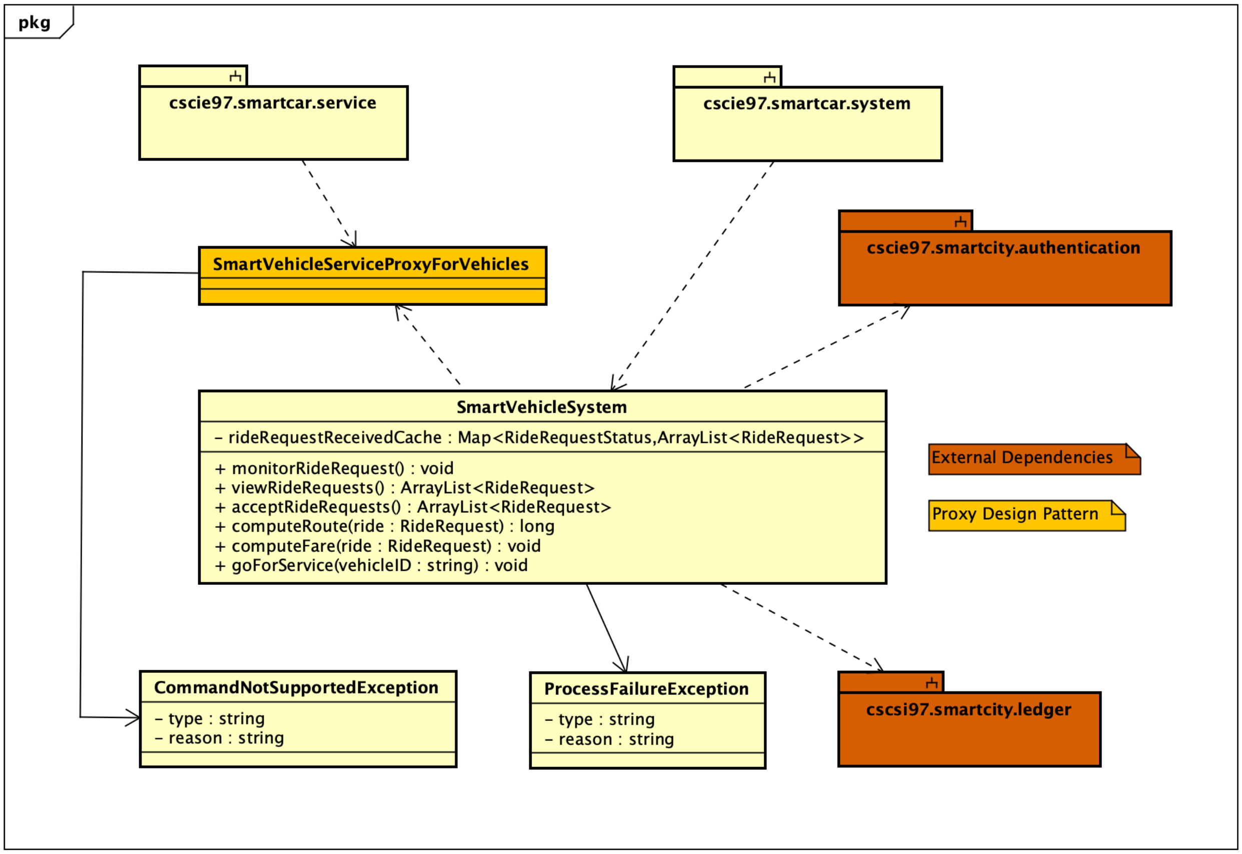
Use Cases



Implementation

This section of the document will describe the implementation details for the Smart Vehicle System.

Class Diagram


pkg
cscie97.smartcar.service
cscie97.smartcar.system
cscie97.smartcity.authentication
SmartVehicleServiceProxyForVehicles
SmartVehicleSystem
− rideRequestReceivedCache : Map<RideRequestStatus,ArrayList<RideRequest>>
+ monitorRideRequest() : void
+ viewRideRequests() : ArrayList<RideRequest>
+ acceptRideRequests() : ArrayList<RideRequest>
+ computeRoute(ride : RideRequest) : long
+ computeFare(ride : RideRequest) : void
+ goForService(vehicleID : string) : void
External Dependencies
Proxy Design Pattern
CommandNotSupportedException
− type : string
− reason : string
ProcessFailureException
− type : string
− reason : string
cscsi97.smartcity.ledger

Class Dictionary

This section specifies the classes for the given package.

SmartVehicleSystem Class

This class is responsible for processing ride requests and the actual movement of people and objects.

Methods

Method Name	Signature	Description
monitorRideRequest	monitorRideRequest() : void	Pull in any new ride requests
viewRideRequests	viewRideRequests() : ArrayList<RideRequest>	View ride requests
acceptRideRequests	acceptRideRequests() : ArrayList<RideRequest>	If possible, accept first valid ride request. If invalid, continue to the next request.

computeRoute	computeRoute(ride : RideRequest) : List<Location>	Determine route necessary to complete trip. Throw an exception if this cannot be done.
computeFare	computeFare(ride : RideRequest) : void	Compute the fare of the trip and charge the associated Rider. If the payment is invalid, throw an exception.
goForService	goForService(vehicleID : string) : void	Place all status-based components to good standing, and maximum fuel level.

Properties

Method Name	Signature	Description
rideRequestReceivedCache	rideRequestReceivedCache : Map<RideRequestStatus,ArrayList<RideRequest>>	In-memory cache of shared ride requests

ProcessFailureException Class

This exception should be thrown in response to any exception throw by a class outside of the smart transportation system.

Properties

Method Name	Signature	Description
type	type : string	The type of the exception
reason	reason : string	The cause of the exception

SmartVehicleServiceProxyForVehicles Interface

This is the implementation of the SmartVehicleService class.

Methods

Method Name	Signature	Description
createRiderProfile	createRiderProfile(customer:RiderProfile):void	Not supported
updateRiderProfile	updateRiderProfile(riderID:string):void	Not supported
monitorVehicles	monitorVehicles():void	Prints out list of Vehicles in the system

createVehicleProfile	createVehicleProfile(vehicleID:string):void	Not supported
updateVehicleProfile	updateVehicleProfile(vehicleID:string):void	Updates an existing Vehicle Profile, and throws an Exception otherwise
shareRideRequests	shareRideRequests():List<RideRequest>	Not supported
shareVehicleReviews	shareVehicleReviews():List<CustomerReview>	Not supported
shareRiderReviews	shareRiderReviews():List<CustomerReview>	Not supported
reportLocationStatus	reportLocationStatus():Map<string,Location>	Update the location of Rider or Vehicle

Properties

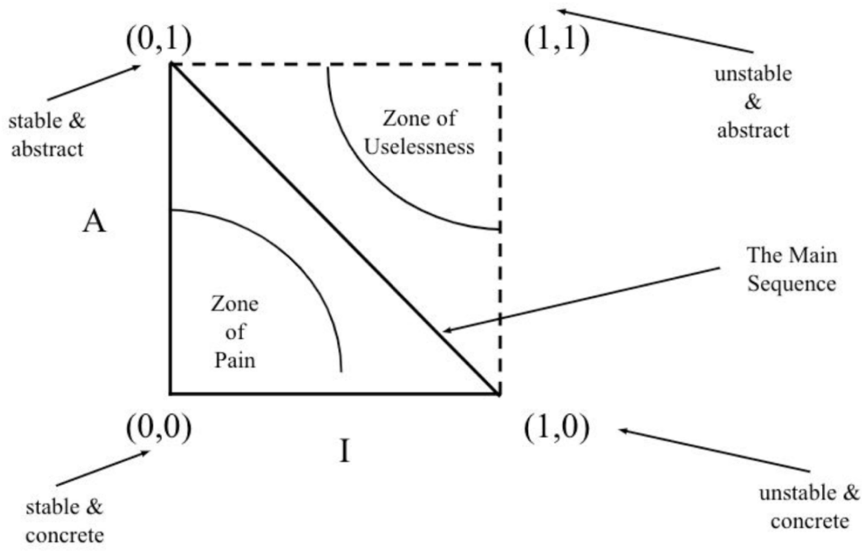
Method Name	**Signature**	**Description**
customers	Customers:Map<string,RiderProfile>	In-memory cache of Rider Profiles within the system
vehicles	Vehicles: Map<string,VehicleProfile>	In-memory cache of Vehicle Profiles within the system

Associations

Method Name	**Signature**	**Description**
openRideRequests	openRideRequests:List<RideRequest>	In-memory cache of new Ride Requests within the system

Implementation Details

The implementation details are provided below.



Instability

Let C_a be the number of classes outside of a package that depend on one or more classes inside the package. These are called afferent classes

Let C_e be the number of classes that one or more of the classes within the given package depend on. These are called efferent classes:

$$\boldsymbol{I = \frac{C_e}{C_e + C_a}}$$

The instability of this class is computed by the following:

$$\boldsymbol{I = \frac{(C_{e_{service}})}{(C_{e_{service}}) + 0} = 1}$$

Abstraction

Let N_c be the number of classes and interfaces in the package

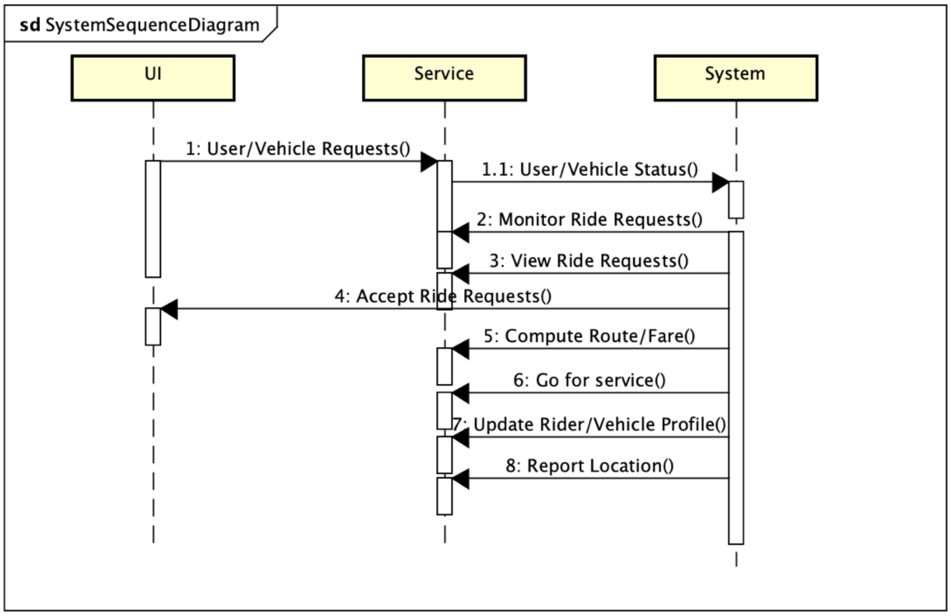
Let N_a be the number of the of non-instantiable classes and interfaces:

$$\boldsymbol{A = \frac{N_a}{N_c}}$$

The abstractness of this class is computed by the following:

$$\boldsymbol{A = \frac{0}{(N_{c_{system}} + N_{c_{exception}})} = 0}$$

This would suggest that this package is unstable and concrete. The sequence diagram below displays the sequence of events by the utilization of this package.


sd SystemSequenceDiagram
UI
Service
System
1: User/Vehicle Requests()
1.1: User/Vehicle Status()
2: Monitor Ride Requests()
3: View Ride Requests()
4: Accept Ride Requests()
5: Compute Route/Fare()
6: Go for service()
7: Update Rider/Vehicle Profile()
8: Report Location()

Exception Handling

While other exceptions exist as a result of the other services, the primary exceptions handled with respect to the this package are:

- CommandNotSupportedException
- ProcessFailureException

The **CommandNotSupportedException** accounts for any requested action performed by an individual with invalid access rights (*i.e., riders who attempt to create a vehicle profile*).

The **ProcessFailureException** accounts for any failure within the system itself. Many examples can trigger this exception (*i.e., no vehicles existing for a specific set of specifications, no vehicles existing at all, failed payment process, etc*.).

Exceptions captured from external services should be compartmentalized and provided through the ProcessFailureException.

ADMINISTRATION CONSOLE (UI)

Introduction

This section defines the design for the Smart Vehicle Administration Console.

Overview

The Administration console serves as the main entry point for an administrative user. It provides options that assist in the management and monitoring of the Smart Transportation System. In addition to creating new vehicle profiles, it can monitor both them and those of the customers as well. In any instance, only the administrator is aware of all the ratings and history of every vehicle/customer registered.

Requirements

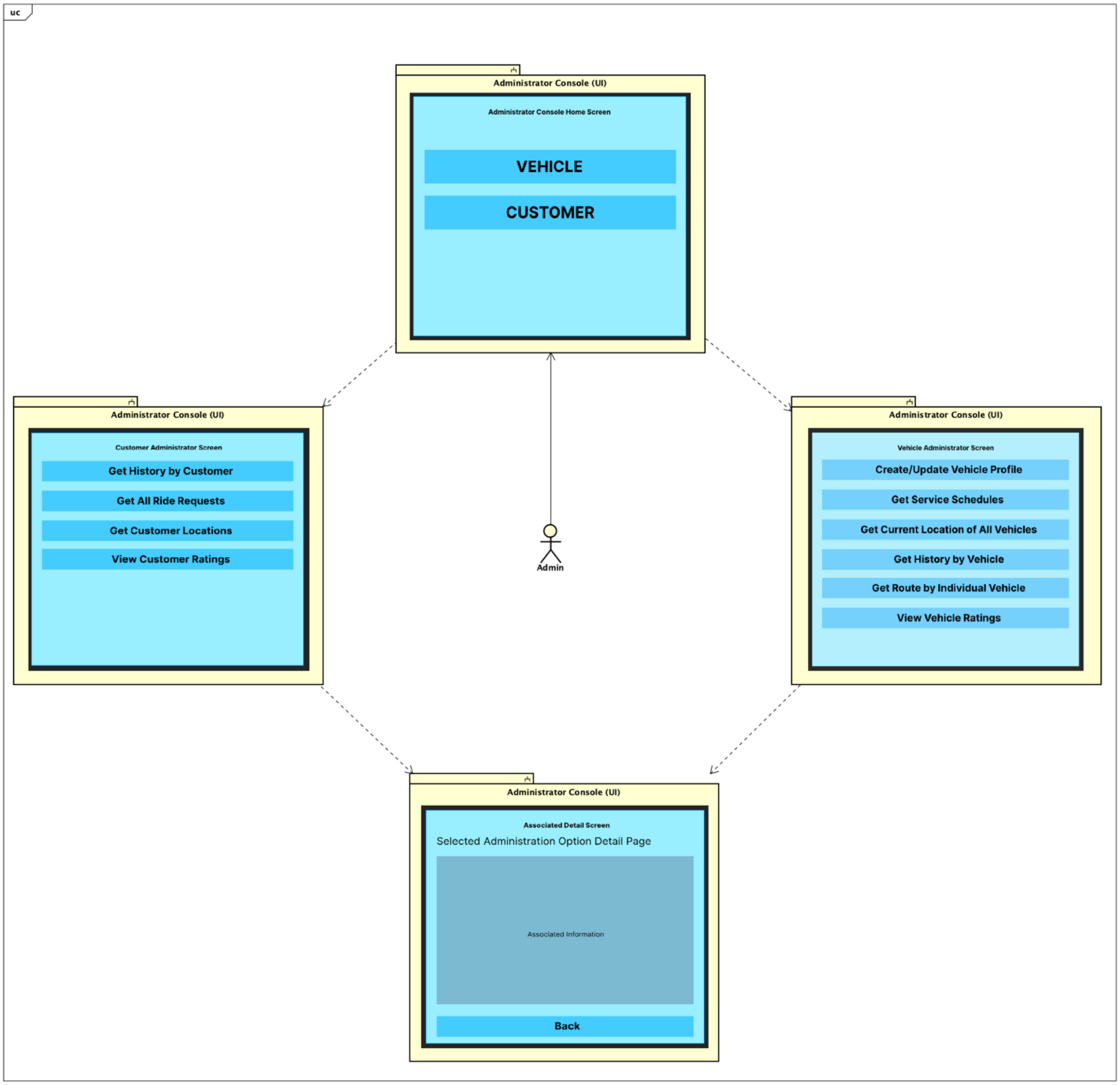
This section provides a brief summary of the requirements for the Administration Console.

The Smart Vehicle Service is responsible for assuring that all vehicles and users are monitored such that all of their needs, with respect to the system, are addressed. The console allows the administrator to know the current location, status, and performance of each.

The Administration Console utilizes the following entities:

- Views
    - Vehicle
    - Customers
    - Ride Requests

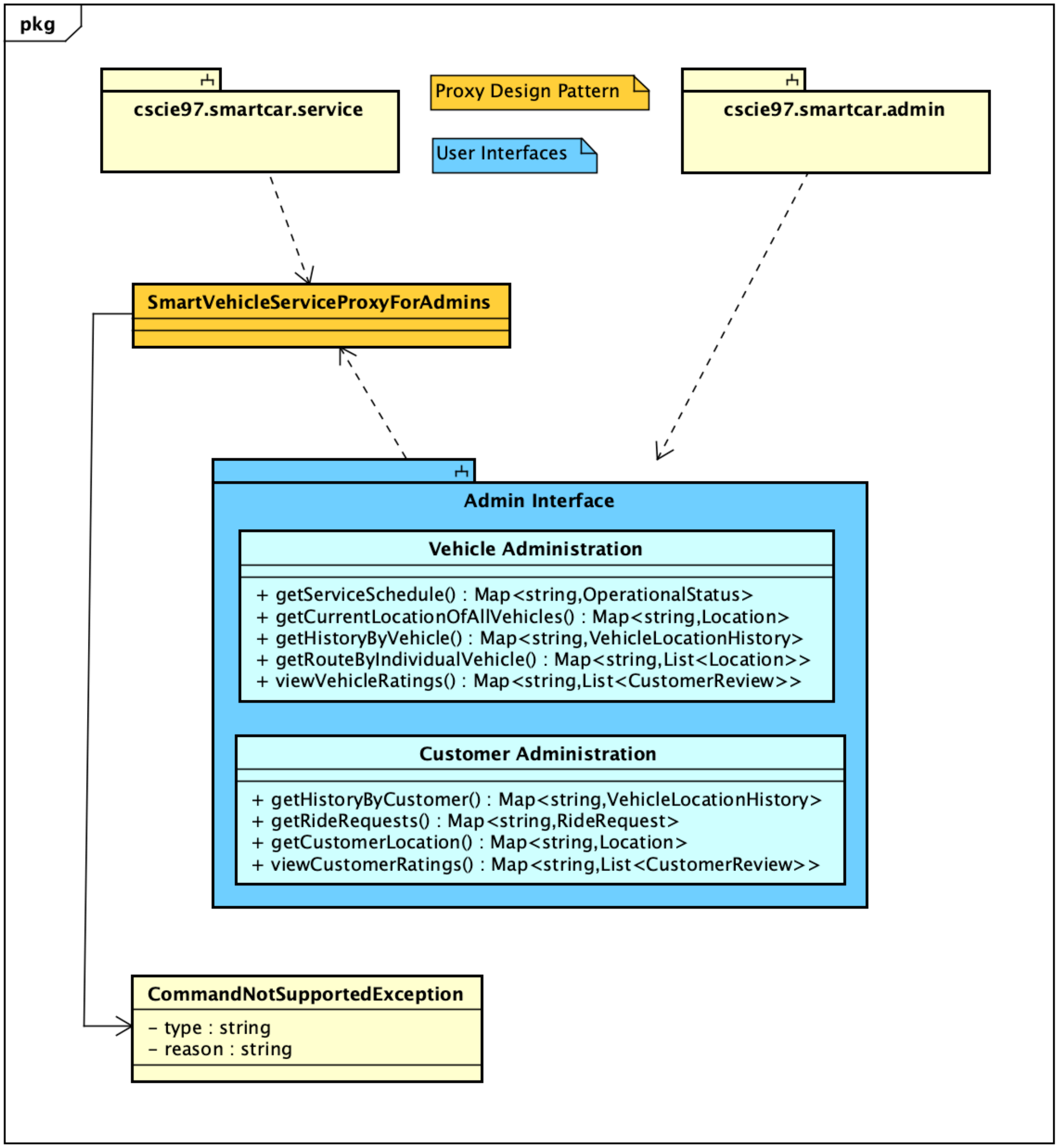
Use Cases


uc
Administrator Console (UI)
Administrator Console Home Screen
VEHICLE
CUSTOMER
Administrator Console (UI)
Customer Administrator Screen
Get History by Customer
Get All Ride Requests
Get Customer Locations
View Customer Ratings
Admin
Administrator Console (UI)
Vehicle Administrator Screen
Create/Update Vehicle Profile
Get Service Schedules
Get Current Location of All Vehicles
Get History by Vehicle
Get Route by Individual Vehicle
View Vehicle Ratings
Administrator Console (UI)
Associated Detail Screen
Selected Administration Option Detail Page
Associated Information
Back

Implementation

This section of the document will describe the implementation details for the Administration Console.

Class Diagram


pkg
cscie97.smartcar.service
Proxy Design Pattern
User Interfaces
cscie97.smartcar.admin
SmartVehicleServiceProxyForAdmins
Admin Interface
Vehicle Administration
+ getServiceSchedule() : Map<string,OperationalStatus>
+ getCurrentLocationOfAllVehicles() : Map<string,Location>
+ getHistoryByVehicle() : Map<string,VehicleLocationHistory>
+ getRouteByIndividualVehicle() : Map<string,List<Location>>
+ viewVehicleRatings() : Map<string,List<CustomerReview>>
Customer Administration
+ getHistoryByCustomer() : Map<string,VehicleLocationHistory>
+ getRideRequests() : Map<string,RideRequest>
+ getCustomerLocation() : Map<string,Location>
+ viewCustomerRatings() : Map<string,List<CustomerReview>>
CommandNotSupportedException
- type : string
- reason : string

Class Dictionary

This section specifies the classes for the given package.

Administrator Interface

This is the primary interface for an administrative user.

Associations

Method Name	Signature	Description
vehicleUI	vehicleUI:VehicleAdministration	The vehicle management view
customerUI	customerUI:CustomerAdministration	The customer management view

VehicleAdministration Class

This is an administrative view for Vehicle management.

Properties

Method Name	Signature	Description
getServiceSchedule	getServiceSchedule() : Map<string,OperationalStatus>	A Map of vehicles and their current service status
getCurrentLocationOfAllVehicles	getCurrentLocationOfAllVehicles() : Map<string,Location>	Return location of all vehicles
getHistoryByVehicle	getHistoryByVehicle() : Map<string,VehicleLocationHistory>	Return all history objects of a vehicle
getRouteByIndividualVehicle	getRouteByIndividualVehicle() : Map<string,List<Location>>	Return all routes a vehicle
viewVehicleRatings	viewVehicleRatings() : Map<string,List<CustomerReview>>	Returns the average rating the vehicle

CustomerAdministration Class

This is an administrative view for Customer Management.

Methods

Method Name	Signature	Description
getHistoryByCustomer	getHistoryByCustomer() : Map<string,VehicleLocationHistory>	Return all history objects of a customer

getRideRequests	getRideRequests() : Map<string,RideRequest>	Get all active ride requests
getCustomerLocation	getCustomerLocation() : Map<string,Location>	Get location of each customer
viewCustomerRatings	viewCustomerRatings() : Map<string,List<CustomerReview>>	View customer ratings

SmartVehicleServiceProxyForAdmins Class

This is the implementation of the SmartVehicleService class.

Methods

Method Name	**Signature**	**Description**
createRiderProfile	createRiderProfile(customer:RiderProfile):void	Not supported
updateRiderProfile	updateRiderProfile(riderID:string):void	Not supported
monitorVehicles	monitorVehicles():void	Prints out list of Vehicles in the system
createVehicleProfile	createVehicleProfile(vehicleID:string):void	Creates a new Vehicle Profile
updateVehicleProfile	updateVehicleProfile(vehicleID:string):void	Updates an existing Vehicle Profile, and throws an Exception otherwise
shareRideRequests	shareRideRequests():List<RideRequest>	Send out new ride request list
shareVehicleReviews	shareVehicleReviews():List<CustomerReview>	Send out vehicle reviews list
shareRiderReviews	shareRiderReviews():List<CustomerReview>	Send out rider reviews list
reportLocationStatus	reportLocationStatus():Map<string,Location>	Update the location of Rider or Vehicle

Properties

Method Name	**Signature**	**Description**
customers	Customers:Map<string,RiderProfile>	In-memory cache of Rider Profiles within the system

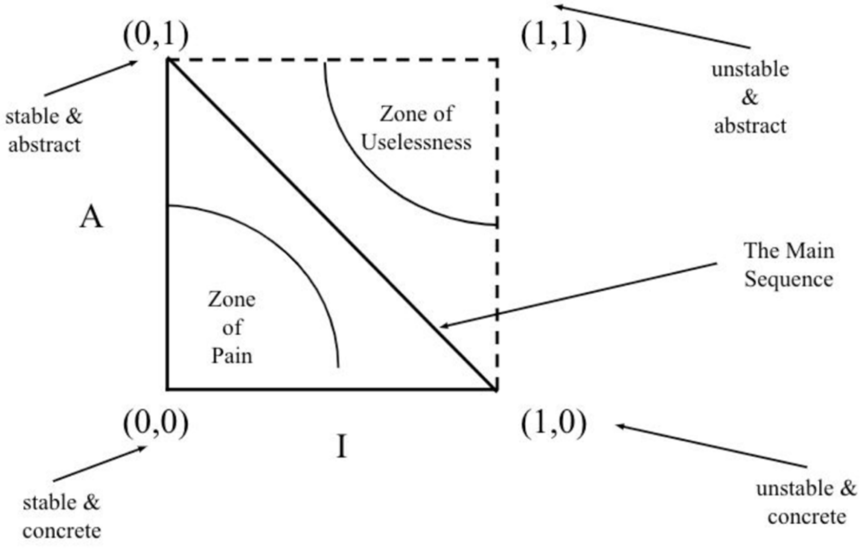
vehicles	Vehicles: Map<string,VehicleProfile>	In-memory cache of Vehicle Profiles within the system

Associations

Method Name	**Signature**	**Description**
openRideRequests	openRideRequests:List<RideRequest>	In-memory cache of new Ride Requests within the system

Implementation Details

The implementation details are provided below.



Instability

Let C_a be the number of classes outside of a package that depend on one or more classes inside the package. These are called afferent classes

Let C_e be the number of classes that one or more of the classes within the given package depend on. These are called efferent classes:

$$\boldsymbol{I = \frac{C_e}{C_e + C_a}}$$

The instability of this class is computed by the following:

$$\boldsymbol{I = \frac{0}{0 + (C_{a_{service}})} = 0}$$

Abstraction

Let N_c be the number of classes and interfaces in the package

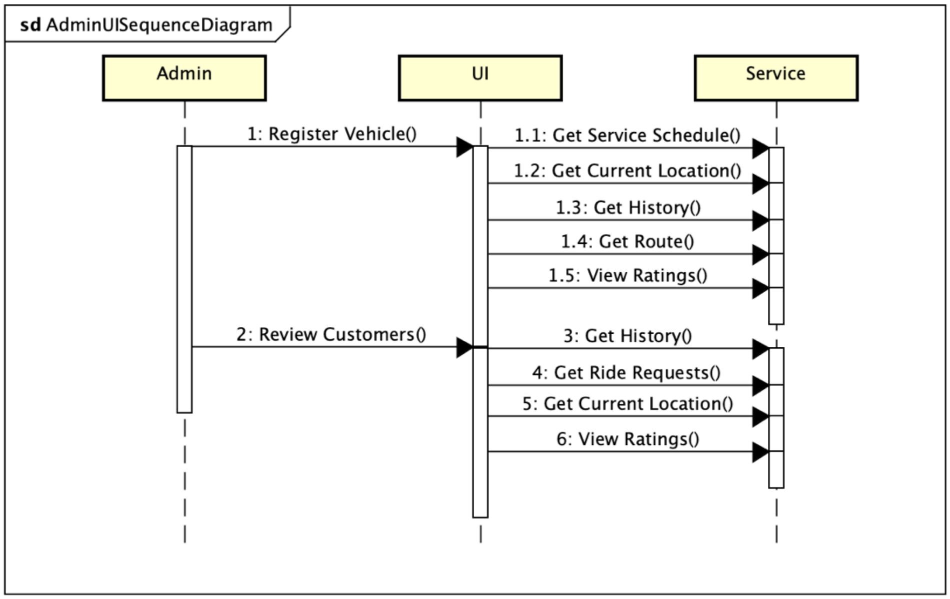
Let N_a be the number of the of non-instantiable classes and interfaces:

$$\boldsymbol{A = \frac{N_a}{N_c}}$$

The abstractness of this class is computed by the following:

$$\boldsymbol{A = \frac{0}{(N_{c_{admin}})} = 0}$$

This would suggest that this package is stable and concrete. The sequence diagram below displays the sequence of events by the utilization of this package.



Exception Handling

While other exceptions exist as a result of the other services, the primary exceptions handled with respect to the this package are:

- CommandNotSupportedException
- ProcessFailureException

The **CommandNotSupportedException** accounts for any requested action performed by an individual with invalid access rights (*i.e., riders who attempt to create a vehicle profile*).

MOBILE APPLICATION (UI)

Introduction

This section defines the design for the Mobile Application Console.

Overview

The Mobile Application interface, unlike the Administrator interface, only allows creation and management of elements with respect to itself. That is, a user can only create and manage ride requests, listen to elements related to those requests, and view their own processed invoices. This interface serves as a user's gateway to particular services offered through the other services, and upon ride request completion, the interface updates accordingly.

Requirements

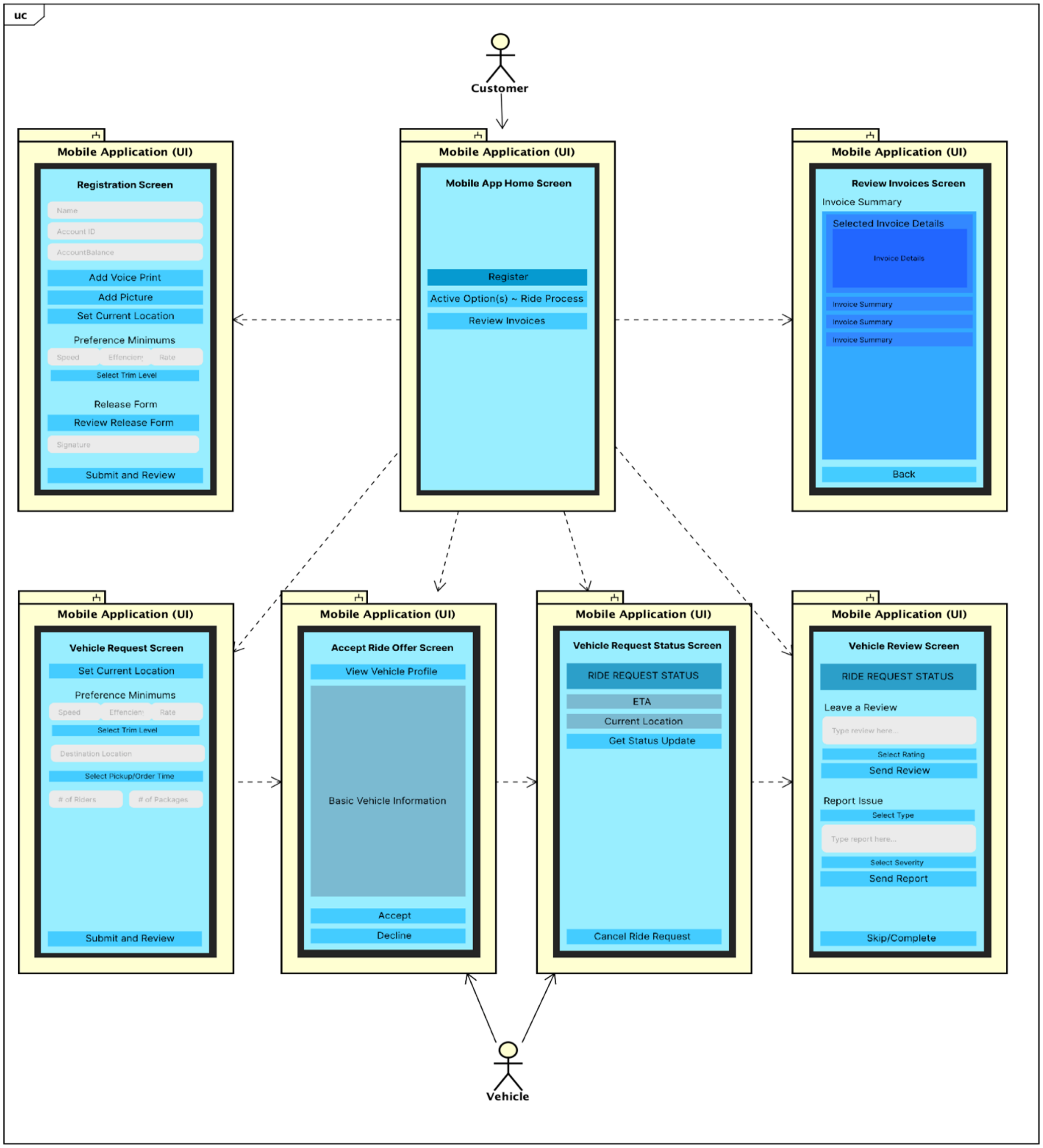
This section provides a brief summary of the requirements for the Mobile Application Interface.

The Mobile Application Interface is responsible for assuring that users can register and request rides from existing vehicles base on a set of criteria that they define. They can manage the life cycle of a ride request and view the invoices of closed requests.

The Mobile Application Interface utilizes the following entities:

- Views
  - Registration Form
  - Vehicle Request Form
  - Accept Ride Offer
  - Vehicle Request Status
  - Vehicle Review Form
  - Review Invoices

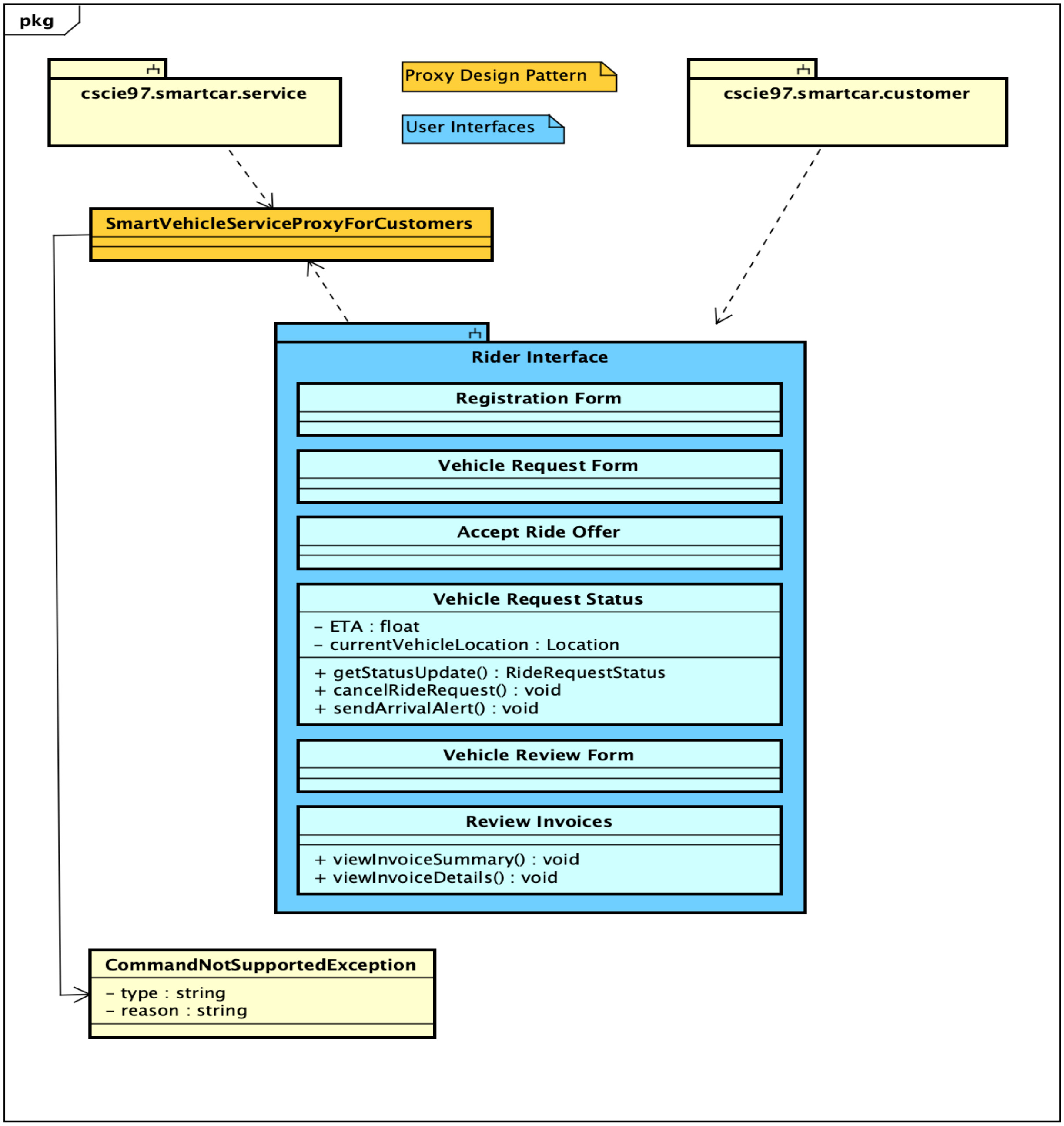
Use Cases


uc
Customer
Mobile Application (UI)
Registration Screen
Name
Account ID
AccountBalance
Add Voice Print
Add Picture
Set Current Location
Preference Minimums
Speed
Effenciency
Rate
Select Trim Level
Release Form
Review Release Form
Signature
Submit and Review
Mobile Application (UI)
Mobile App Home Screen
Register
Active Option(s) ~ Ride Process
Review Invoices
Mobile Application (UI)
Review Invoices Screen
Invoice Summary
Selected Invoice Details
Invoice Details
Invoice Summary
Invoice Summary
Invoice Summary
Back
Mobile Application (UI)
Vehicle Request Screen
Set Current Location
Preference Minimums
Speed
Effenciency
Rate
Select Trim Level
Destination Location
Select Pickup/Order Time
# of Riders
# of Packages
Submit and Review
Mobile Application (UI)
Accept Ride Offer Screen
View Vehicle Profile
Basic Vehicle Information
Accept
Decline
Mobile Application (UI)
Vehicle Request Status Screen
RIDE REQUEST STATUS
ETA
Current Location
Get Status Update
Cancel Ride Request
Mobile Application (UI)
Vehicle Review Screen
RIDE REQUEST STATUS
Leave a Review
Type review here...
Select Rating
Send Review
Report Issue
Select Type
Type report here...
Select Severity
Send Report
Skip/Complete
Vehicle

Implementation

This section of the document will describe the implementation details for the Mobile Application Interface.

Class Diagram


pkg
cscie97.smartcar.service
Proxy Design Pattern
User Interfaces
cscie97.smartcar.customer
SmartVehicleServiceProxyForCustomers
Rider Interface
Registration Form
Vehicle Request Form
Accept Ride Offer
Vehicle Request Status
– ETA : float
– currentVehicleLocation : Location
+ getStatusUpdate() : RideRequestStatus
+ cancelRideRequest() : void
+ sendArrivalAlert() : void
Vehicle Review Form
Review Invoices
+ viewInvoiceSummary() : void
+ viewInvoiceDetails() : void
CommandNotSupportedException
– type : string
– reason : string

Class Dictionary

This section specifies the classes for the given package.

Rider Interface

This is the primary view for customers within the system.

Associations

Method Name	Signature	Description
registrationForm	registrationForm:RegistrationForm	The registration form view
vehicleRequestForm	vehicleRequestForm: VehicleRequestForm	The vehicle request form view
acceptRideOffer	acceptRideOffer: AcceptRideOffer	The accept ride offer view
vehicleRequestStatus	vehicleRequestStatus: VehicleRequestStatus	The vehicle request status view
vehicleReviewForm	vehicleReviewForm:VehicleReviewForm	The vehicle request view
reviewInvoices	reviewInvoices:ReviewInvoices	The review invoices view

RegistrationForm Class

This view allows the user to allow vehicles to get permission to communicate and offer ride requests to it.

Associations

Method Name	Signature	Description
smartVehicleServiceProxyForCustomer	smartVehicleServiceProxyForCustomer: SmartVehicleServiceProxyForCustomer	Proxy command from the SmartVehicle Service

VehicleRequestForm Class

This view allows the user to request rides from vehicles within the city.

Associations

Method Name	Signature	Description
smartVehicleServiceProxyForCustomer	smartVehicleServiceProxyForCustomer: SmartVehicleServiceProxyForCustomer	Proxy command from the SmartVehicle Service

AcceptRideOffer Class

This view allows users to accept incoming ride offers from vehicles that fulfill the criteria of their ride requests.

Associations

Method Name	Signature	Description
smartVehicleServiceProxyForCustomer	smartVehicleServiceProxyForCustomer: SmartVehicleServiceProxyForCustomer	Proxy command from the SmartVehicle Service

VehicleRequestStatus Class

This view allows for the monitoring of the final acceptance process after accepting a ride offer.

Associations

Method Name	Signature	Description
smartVehicleServiceProxyForCustomer	smartVehicleServiceProxyForCustomer: SmartVehicleServiceProxyForCustomer	Proxy command from the SmartVehicle Service

VehicleReviewForm Class

This view allows the user to leave reviews and reports based on their ride experiences.

Associations

Method Name	Signature	Description
smartVehicleServiceProxyForCustomer	smartVehicleServiceProxyForCustomer: SmartVehicleServiceProxyForCustomer	Proxy command from the

		SmartVehicle Service

ReviewInvoices Class

This view allows users to review processed invoices.

Methods

Method Name	**Signature**	**Description**
getStatusUpdate	getStatusUpdate() : RideRequestStatus	Get the status of the RideRequestStatus
cancelRideRequest	cancelRideRequest() : void	Cancel ride request
sendArrivalAlert	sendArrivalAlert() : void	End the trip

Properties

Method Name	**Signature**	**Description**
ETA	ETA : float	Estimated Time of Arrival
currentVehicleLocation	currentVehicleLocation : Location	Current location of the vehicle

Associations

Method Name	**Signature**	**Description**
smartVehicleServiceProxyForCustomer	smartVehicleServiceProxyForCustomer: SmartVehicleServiceProxyForCustomer	Proxy command from the SmartVehicle Service

SmartVehicleServiceProxyForCustomer Class

This is the implementation of the SmartVehicleService class.

Methods

Method Name	**Signature**	**Description**
createRiderProfile	createRiderProfile(customer:RiderProfile):void	Creates a new Rider Profile
updateRiderProfile	updateRiderProfile(riderID:string):void	Updates an existing Rider Profile, and throws an Exception otherwise

monitorVehicles	monitorVehicles():void	Not supported
createVehicleProfile	createVehicleProfile(vehicleID:string):void	Not supported
updateVehicleProfile	updateVehicleProfile(vehicleID:string):void	Not supported
shareRideRequests	shareRideRequests():List<RideRequest>	Send out new ride request list
shareVehicleReviews	shareVehicleReviews():List<CustomerReview>	Send out vehicle reviews list
shareRiderReviews	shareRiderReviews():List<CustomerReview>	Send out rider reviews list
reportLocationStatus	reportLocationStatus():Map<string,Location>	Update the location of Rider or Vehicle

Properties

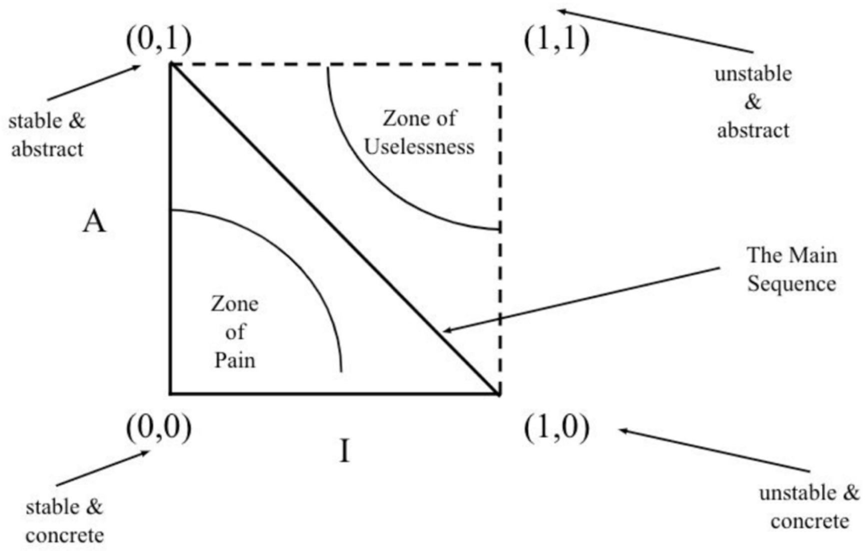
Method Name	**Signature**	**Description**
customers	Customers:Map<string,RiderProfile>	In-memory cache of Rider Profiles within the system
vehicles	Vehicles: Map<string,VehicleProfile>	In-memory cache of Vehicle Profiles within the system

Associations

Method Name	**Signature**	**Description**
openRideRequests	openRideRequests:List<RideRequest>	In-memory cache of new Ride Requests within the system

Implementation Details

The implementation details are provided below.



Instability

Let C_a be the number of classes outside of a package that depend on one or more classes inside the package. These are called afferent classes

Let C_e be the number of classes that one or more of the classes within the given package depend on. These are called efferent classes:

$$\boldsymbol{I = \frac{C_e}{C_e + C_a}}$$

The instability of this class is computed by the following:

$$\boldsymbol{I = \frac{0}{0 + (C_{a_{service}})} = 0}$$

Abstraction

Let N_c be the number of classes and interfaces in the package

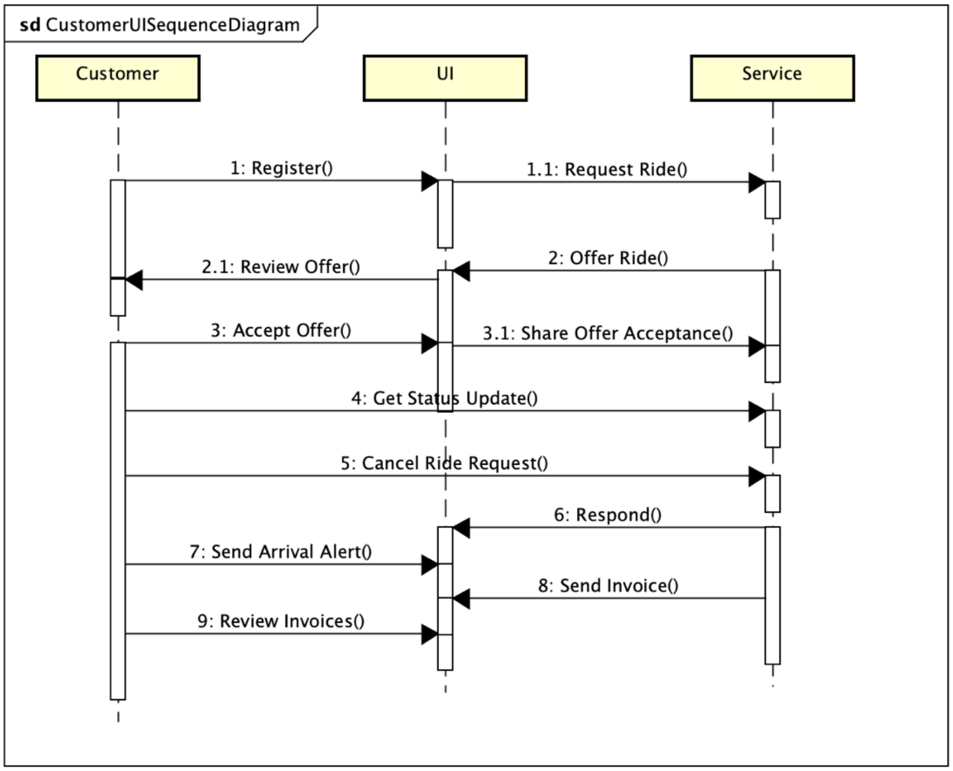
Let N_a be the number of the of non-instantiable classes and interfaces:

$$A = \frac{N_a}{N_c}$$

The abstractness of this class is computed by the following:

$$A = \frac{0}{(N_{c_{rider}})} = 0$$

This would suggest that this system is stable and concrete. The sequence diagram below displays the sequence of events by the utilization of this package.



Exception Handling

While other exceptions exist as a result of the other services, the primary exceptions handled with respect to the this package are:

- CommandNotSupportedException
- ProcessFailureException

The **CommandNotSupportedException** accounts for any requested action performed by an individual with invalid access rights (*i.e., riders who attempt to create a vehicle profile*).

RESULTS DOCUMENT

1) **Did the modular approach to the design help?**
    a. Yes, the modular approach helped organize the different stages of the ride request life cycle, in such a way that it made it easier to allow the subcomponents of this application to work together instead of directly tying them all together.

2) **Once you had the high-level architecture defined, were you able to design each of the modules sequentially?**
    a. Yes, I was. The only changes that were made out of order was the introduction of the Command Pattern with respect to the interfaces.

3) **Did the reuse of the Authentication Service and Ledger Service help or hinder your design?**
    a. The use of these services helped because the both safeguard the system by limiting those who can create and modify vehicle profiles, and assure that the distribution and movement of payment units reflects that of a real-world application.

4) **What design patterns did you apply? Did you find that they helped simplify the design?**
    a. Command
        i. In combination with the proxy pattern, I used this pattern to handle the events within the extensions of the SmartVehicleService abstract class to access specific commands.
    b. Proxy
        i. In combination with the command pattern, I used this pattern to handle the events within the extensions of the SmartVehicleService abstract class to access specific commands. However, this was to further safeguard the application by ensuring that the extensions assured that some features were no longer supported.
    c. Singleton
        i. This design pattern was used solely for the purpose of storing in-memory caches of the different stages of the ride request process. It serves as an intermediary layer between the application and the data, and it prevents multiple instances of the unique entity.

5) **Were you able to brainstorm with your peer review team?**
    a. Yes, we brainstormed for a large portion of the assignment.

6) **Did the peer review help improve your design?**
    a. Yes, we had multiple review sessions and our designs were updated accordingly each time.

7) **Do you think you could implement this design?**
    a. Yes, I think I can complete a functioning implementation of this design. However, I'm also sure it could be decoupled even further. Also, my in-memory cache may not be necessary if I'm 100% sure I have an external database available.

8) **Comments from peer design review.**
    a. "Your initial class diagram is very hard to read. Make sure you include others in your following sections breaking this up a bit."
    b. "Your design opens up the opportunity to create an abstract class called Profiles so you don't have to recreate the same properties more than once."
    c. "You are creating new Vehicles and then using the other services to monitor them, but I think you can just create those vehicles directly within the other services."
    d. "You don't actually have to show the CommandNotSupportedException in every diagram"