Smart city model service design document

Date: 09/26/20

Author: Yuri Machkasov

Reviewer: Eric Laribee, Licheng Xu

Contents

Introduction:

The Smart City Model Service is responsible for maintaining the state of Internet of Things (IoT) devices within the Smart City. The IoT Devices are designed to support the needs of city residents and guests. All IoT devices contain sensors that are able to collect and share data.

This design specifies the implementation of a service that abstracts the relevant properties of the Smart Cities, their residents and visitors, and individual IoT devices, and allows for definition and manipulation of the corresponding objects, which serve as a proxy to the physical device, through an API. The service also represents the system of record for the state of all domain objects.

Requirements:

The service will be responsible for managing and manipulating the state of the City domain objects. For each of the three groups of objects (the cities, the people and the physical IoT-connected smart devices) it will define APIs which will enable the user of the service to add those objects to the model instance and change their properties. These APIs will be the only way for any entity to interact with the service. To use any of these interfaces, the client must be properly authenticated; the resulting token must be passed as an argument to all calls.

The interface will be exposed as an ability for the service to process and execute text commands. The commands will include:

1. Defining, accessing and updating the City configuration

2. Creating/Simulating sensor events and output feedback through device speakers

3. Accessing the state of IoT devices and sending command messages to them

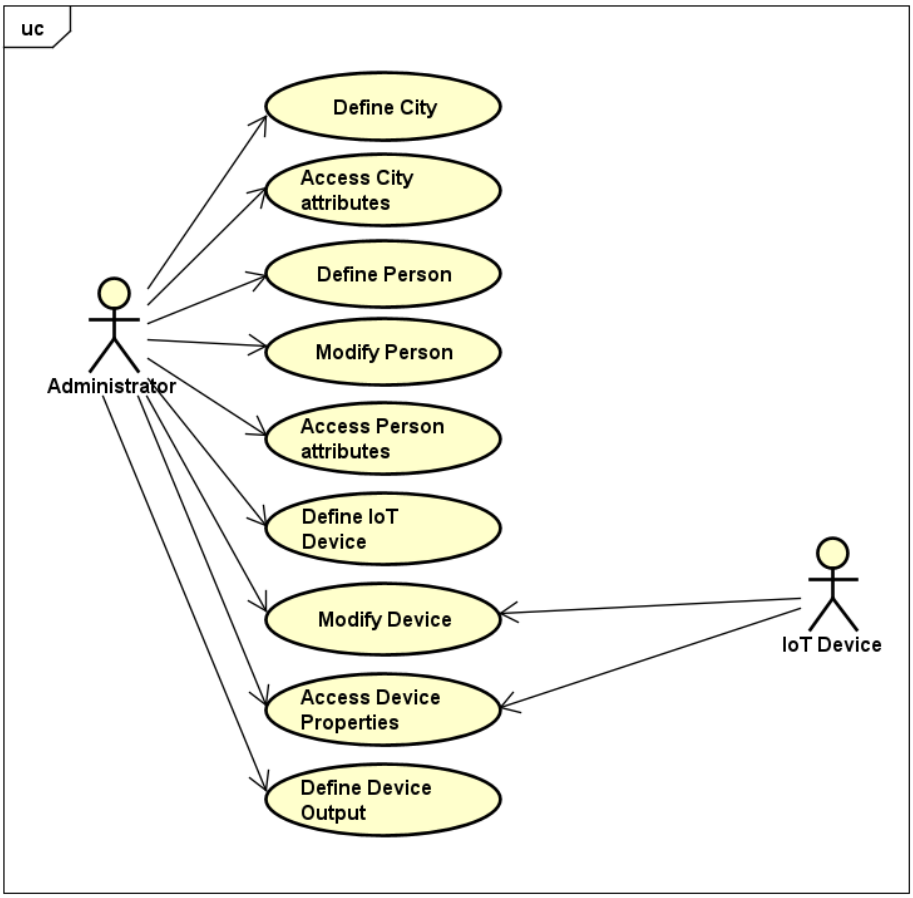
4. Accessing the IoT events

5. Monitoring and supporting Residents and Visitors

For more details please refer to the requirements document

Use Cases:

The service, as part of the overall Smart City architecture, is concerned with two major use case categories: the interactions between the City administrator and all objects encompassed by the model, and the interactions between the IoT devices and the objects in the model that represent their abstractions. Correspondingly, there are two actors for this service. Please refer to the following diagram to determine the implemented use cases:



The details of the interactions between persons and devices are not in the scope of this document.

The actors are the Administrator of the service and the physical IoT devices that are present in the city.

Administrators are responsible for managing the collections of City, Person and Device objects within the service. They are able to create and list the objects, and also modify their attributes. We assume that the Administrators negotiate with the service to obtain an authorization token which will be required for any of the operations; the details of the authorization component are outside the scope of this document.

Each IoT device can access and modify its own attributes, and also emit events. The event handling is outside the scope of this document.

These use cases cover all requirements outlined in the previous section.

Use cases:

Define City:

Defining a new City requires an initialized City object and performs validation of its attributes. If it succeeds, a new City is added to the model.

Access City:

The City objects are referenced by their identifiers. The service allows accessing them through this identifier and listing their attributes.

Define Person:

Defining a new Person object requires an initialized object of one of the subtypes of the Person type (Visitor or Resident) and performs validation of its attributes. If it succeeds, a new Person is added to the model.

Modify Person:

The Administrators can modify the attributes of a Person object, identified by a string identifier.

Access Person attributes:

The Administrators can access the attributes of a Person object, identified by a string identifier.

Define IoT device:

Defining a new IoT object requires an initialized object of one of the subtypes of the Device type, and performs validation of its attributes. If it succeeds, a new device is added to the model.

Modify IoT device:

The Administrators and the device itself can modify the attributes of a Device object, including emitting (in case of the Device) or simulating (in case of Administrator) device events. The device events and commands will be defined as part of the city controller module.

Access IoT device attributes:

The Administrators and the device itself can access the attributes of a Device object, including the last event emitted by or simulated for this device.

Define device output:

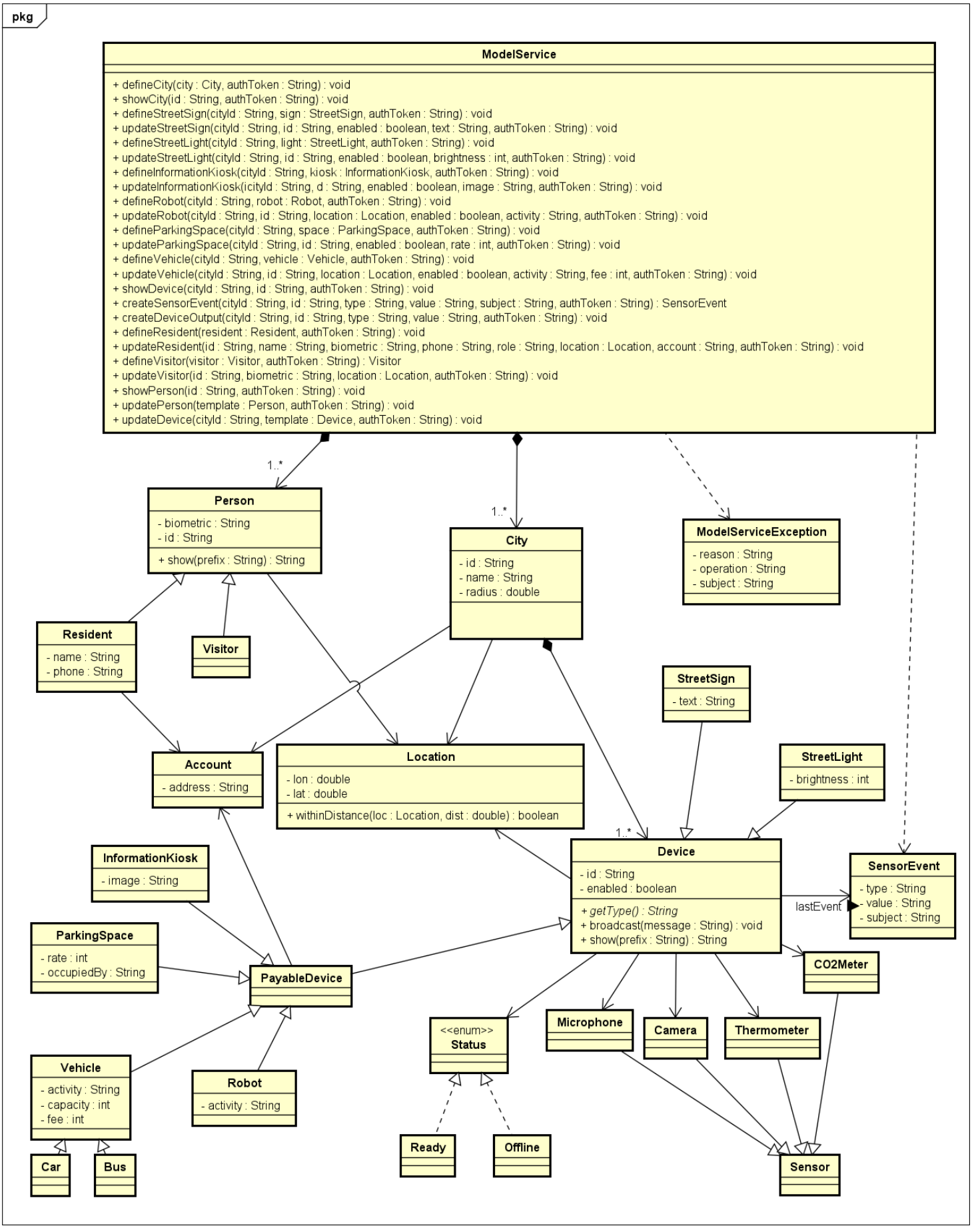
The Administrator can, in response to device events or other considerations, cause a device to broadcast a message through its internal speaker.

Implementation

From the requirements it is easy to see that we will be dealing with classes that abstract three major categories of objects: cities, IoT devices of various types, and people. The latter will form hierarchies according to type. The model service itself, as the system of record, will maintain the one-to-many associations to the cities and people (since they can move freely between the cities), while each city will have a one-to-many relationship to the devices which were defined as belonging to that city. The devices therefore will have unique identifiers within the scope of one city, while other objects will have globally unique identifiers. In case of a device moving outside of a city (temporarily or permanently) their relationship will still be maintained, similar to a car registration in a specific state. The requirement of being able to find people within a city will be addressed by comparing geographical locations.

Class diagram: model service

This diagram represents the model service and its subordinate objects



Class description: model service

ModelService

Associations

Association name	Type	Description
cities	City	A map of known City objects keyed by string identifier
people	Person	A map of known Person objects keyed by string identifier

Methods

These methods mirror the use cases from the previous section and, consequently, fulfill the design requirements. Please note that all of the exposed interfaces require an authentication token; this is not explicitly included in the following signatures for brevity.

Method name	Signature	Description
defineCity	city: City	Performs validation of the incoming City object and adds it to the map
showCity	id: String	Displays information about a City object with the given identifier
defineStreetSign	cityId : String, sign: StreetSign	Performs validation of the incoming StreetSign object and adds it to the map
updateStreetSign	cityId : String, id : String, enabled : boolean, text : String	Allows for modification of the attribute values of a StreetSign with the given id
defineStreetLight	cityId : String, sign: StreetLight	Performs validation of the incoming StreetLight object and adds it to the map
updateStreetLight	cityId : String, id : String, enabled : boolean, brightness: int	Allows for modification of the attribute values of a StreetLight with the given id
defineInformationKiosk	cityId : String, kiosk: InformationKiosk	Performs validation of the incoming InformationKiosk object and adds it to the map
updateInformationKiosk	cityId : String, id : String, enabled : boolean, image: int	Allows for modification of the attribute values of an InformationKiosk with the given id
defineRobot	cityId : String, robot: Robot	Performs validation of the incoming Robot object and adds it to the map
updateRobot	cityId : String, id : String, enabled : boolean, location:Location, activity:String	Allows for modification of the attribute values of a Robot with the given id

defineParkingSpace	cityId : String, space: ParkingSpace	Performs validation of the incoming ParkingSpace object and adds it to the map
updateParkingSpace	cityId : String, id : String, enabled : boolean, rate: int, occupiedBy: String	Allows for modification of the attribute values of a ParkingSpace with the given id
defineVehicle	cityId : String, vehicle: Vehicle	Performs validation of the incoming Vehicle object (should be any of the Vehicle subclasses) and adds it to the map
updateVehicle	cityId : String, id : String, enabled : boolean, location:Location, activity:String, fee: int	Allows for modification of the attribute values of a Vehicle with the given id
showDevice	cityId : String, id : String	Displays current information about a device
defineResident	person: Resident	Performs validation of the incoming Resident object and adds it to the map
updateResident	id : String, name: String, biometric: String, phone: String, role: String, location: Location, account: String	Allows for modification of the attribute values of a Resident with the given id
defineVisitor	person: Visitor	Performs validation of the incoming Visitor object and adds it to the map
updateVisitor	id : String, biometric: String, location: Location	Allows for modification of the attribute values of a Visitor with the given id
showPerson	id : String	Displays current information about a person
createSensorEvent	cityId : String, id : String, type : String, value : String, subject : String	Simulates a sensor event of the given type for the device with this id and installs it as the last event emitted by this device
createDeviceOutput	cityId : String, type : String, value : String	Causes the sensor of the given type (currently only implemented for speakers) to broadcast a message with the given value
updatePerson	template: Person	change the attributes of a Person object with the same identifier as the template to the values specified (that is, non-null) in the template
updateDevice	cityId: String, template: Device	change the attributes of a Device object in this city with the same identifier as the template to the values specified (that is, non-null) in the template

City

The City class represents an abstraction of a smart city. It is characterized by an identifier, and has a blockchain account associated with it for accepting payments. It also contains a collection of devices associated with this city at the moment of their creation. The access to this collection is through an iterator pattern, to lighten the communication protocol and avoid buffering issues.

Properties

Property name	**Type**	**Description**
id	String	A unique identifier
name	String	A common name used to refer to this city
radius	double	This value, in kilometers, represents the extent to which the authority of this city extends (as a circle with the center at the city's coordinates). Can be used to query devices or persons currently within its borders
devices	Device	A map of Device objects defined for this City, keyed by string identifier

Associations

Association name	**Type**	**Description**
location	Location	The coordinates of the geographical center of the City
account	Account	The blockchain account associated with this city

Device

This is the parent class for all IoT devices. It contains an identifier and a flag indicating if the device is enabled, and also has associations with its current location, the mandatory sensors and a status object. Each device also stores the last sensor event it has received.

Properties

Property name	**Type**	**Description**
id	String	A unique identifier (within one city)
enabled	boolean	Whether device is operational

Associations

Association name	**Type**	**Description**
location	Location	The current coordinates of the device
status	Status	The current online status of the device
microphone	Microphone	The onboard sound sensor
camera	Camera	The onboard image sensor
thermometer	Thermometer	The onboard temperature sensor
co2Meter	CO2Meter	The onboard air quality sensor
sensorEvent	SensorEvent	The last event received from onboard sensors or simulated by the administrator

Methods

Method name	Signature	Description
getType		a virtual method that returns the string representation of the device's type
show	prefix: String	returns information about the state of this object, one attribute per line; each line starts with the specified prefix
broadcast	message: String	Placeholder for broadcasting a message through the device's speaker

PayableDevice

This is the extension of the Device object that has an associated blockchain account for accepting payments

Associations

Association name	Type	Description
account	Account	The blockchain account associated with this device

Person

This is an object representing properties common to all persons. It has a unique identifier and also contains biometric information and the current location of the person. One of the subclasses of this class, Visitor, does not have any additional properties or associations.

Properties

Property name	Type	Description
id	String	A unique identifier
biometric	String	The encoded biometric information

Associations

Association name	Type	Description
location	Location	The current coordinates of the person

Methods

Method name	Signature	Description
show	prefix: String	returns information about the state of this object, one attribute per line; each line starts with the specified prefix

Location

This is a container for the coordinates of an object.

Properties

Property name	Type	Description
lat	double	The latitude (in degrees)
lon	double	The longitude (in degrees)

Methods

Method name	Signature	Description
withinDistance	otherLocation: Location, distance: double	Returns a boolean flag that indicates whether the location specified as the parameter is within a given distance (in km) from this location

Account

This object encapsulated the access to a blockchain account; it contains the string address, and the account itself can be looked up by this address in the Ledger service.

Properties

Property name	Type	Description
address	String	The unique address

StreetSign

This class represents a device that displays a changeable text

Properties

Property name	Type	Description
text	String	A text to be displayed on the sign

StreetLight

This class represents a device that has a graduated light emitter

Properties

Property name	Type	Description
brightness	int	The brightness level (in increments; 0 means switched off)

Robot

This class is an abstraction of a robotic device. It is capable of performing changeable activities.

Properties

Property name	Type	Description
activity	String	The current activity

ParkingSpace

This class is an abstraction of a paid parking space..

Properties

Property name	Type	Description
rate	int	The parking rate at this space per hour
occupiedBy	String	The identifier of the vehicle in this place

InformationKiosk

This class is an abstraction of an information device that can present a changeable image.

Property name	Type	Description
image	String	The URL of the current image

Vehicle

This class is a parent for vehicle objects. It contains the capacity of the vehicle and the current fee associated with using it, and also the changeable activity string representing the vehicle's current purpose. The two subclasses of the vehicle, Car and Bus, do not have any additional properties or associations.

Properties

Property name	Type	Description
capacity	int	The person capacity of the vehicle
fee	int	Current fee for using this vehicle
activity	String	The current activity

Resident

This is a subclass of a Person that represents a resident of the city. In addition to the properties and associations of its parent class, it also contains the person's name and phone number, and is associated with a blockchain account.

Properties

Property name	Type	Description
name	String	The person's name
phone	String	The person's phone number

Associations

Association name	Type	Description
account	Account	The blockchain account associated with this person

Sensor

This is the base class for the sensors contained in each IoT device. The functionality of the sensors and devices will be addressed in further design documents. None of the subclasses contain any additional properties or associations.

ModelServiceException

An exception that is being thrown by the service if the requested operation could not be performed. Contains information on both the offending command and the reason for failure

Properties

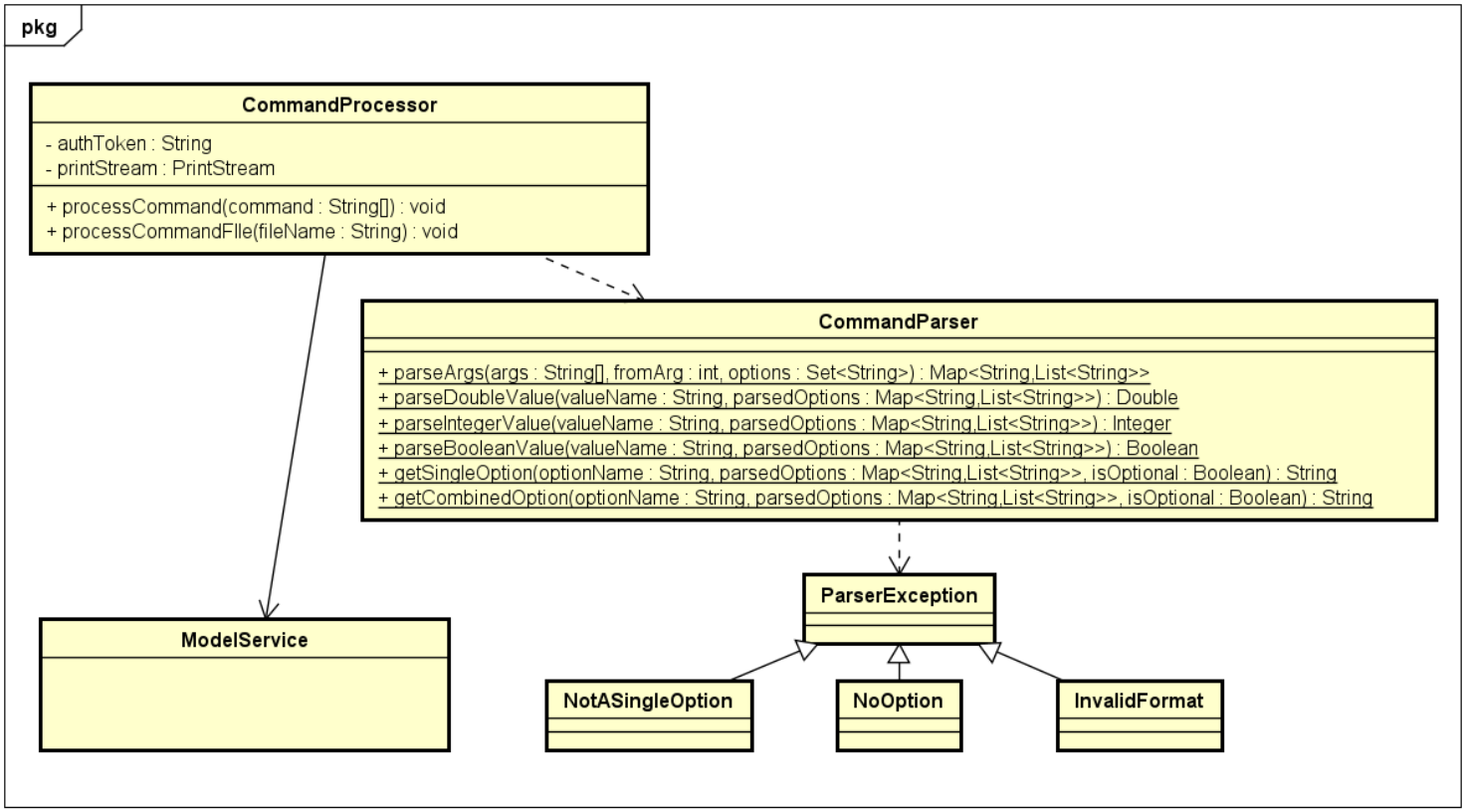
Property name	**Type**	**Description**
reason	String	The text representation of the error
operation	String	The requested operation (create/show/define/update)
subject	String	The type of the object on which the operation was performed

Status

An enumeration having two possible values: **READY** and **OFFLINE**

Class diagram: Command processor

This diagram represents the model service client which is capable of reading text commands and transforming them into API calls into the model service instance.



Class description: command processor

CommandProcessor

This is the main class representing the model service client accepting and executing text commands.

Properties

Property name	Type	Description
authToken	String	The authorization token to be passed to the model service API
printStream	PrintStream	The stream for directing the command output. Defaults to the system output

Methods

Method name	Signature	Description
processCommand	command: String[]	Interprets the arguments as command tokens, parses it, invokes the corresponding model service method and prints out the result of that command. Throws a ModelException if an underlying command was not successful
processCommandFile	fileName: File	Reads the lines in the supplied file, interpreting each as a separate model service command. If a command fails, logs the exception and the line number where the error occurred, and continues processing

CommandParser

A utility class concerned with separating the command string into tokens corresponding to options for a particular command, and extracting values for these options. All methods are static.

Methods

Method name	Signature	Description
parseArgs	command: String[], startIdx: int, options: Set<String>	Reads the command string starting from the indicated position and fills the map of option values. Throws a ParserException if not successful
parseDoubleValue	valueName: String, parsedOptions : Map<String,List<String>>	Tries to extract a double value from the set of option values. Throws a ParserException if not successful
parseIntegerValue	valueName: String, parsedOptions : Map<String,List<String>>	Tries to extract an integervalue from the set of option values. Throws a ParserException if not successful
parseBooleanValue	valueName: String, parsedOptions : Map<String,List<String>>	Tries to extract a double value from the set of option values. Throws a ParserException if not successful
getSingleOption	optionName: String, parsedOptions :	Checks that the set of option values has at most a single value for the specified option and returns it; if the

	Map<String,List<String>>, isOptional: boolean	“isOptional” flag is false, then also verifies that this option is present. Throws a ParserException if conditions are not met
getCombinedOption	optionName: String, parsedOptions : Map<String,List<String>>, isOptional: boolean	Concatenates all values for the specified option into a single string and returns it; if the “isOptional” flag is false, then also verifies that this option is present. Throws a ParserException if conditions are not met

ParserException

A parent class for all exceptions occurring during the parsing process. Does not have any properties or methods in addition to the generic Exception, but sets the exception message. Its subclasses, also empty, correspond to the individual exception causes.

Command syntax

This section describes the syntax of the text commands that are capable of being understood by the processor and passed on to the model service. These commands correspond to the use cases and as such fulfill the service requirements.

City Commands

Define a city, characterized by a unique identifier:

```
define city <city_id> name <name> account <address> lat <float> long <float> radius <float>
```

Show the details of a city. Prints out the attributes of the city object, the associated devices, and also of the person objects whose location falls within the boundaries of the city (as defined by a circle with the center in the city's location and a specified radius):

```
show city <city_id>
```

Device Commands

These commands must start with a command group keyword (define/update/show) and a second keyword specifying the type of the device, followed by the identifier. Note that the identifier string in all cases is a string that contains both the id of the city where the device has been (or will be) first registered and the identifier of the device itself. It is the job of the command processor to separate them and pass correct information to the model service.

Define a street sign

```
define street-sign <city_id>:<device_id> lat <float> long <float> enabled (true|false) text
<text>
```

Update a street sign

```
update street-sign <city_id>:<device_id> [enabled (true|false)] [text <text>]
```

Define an information kiosk

```
define info-kiosk <city_id>:<device_id> lat <float> long <float> enabled (true|false) image <uri>
```

Update an information kiosk

```
update info-kiosk <city_id>:<device_id> [enabled (true|false)] [image <uri>]
```

Define a street light

```
define street-light <city_id>:<device_id> lat <float> long <float> enabled (true|false)
brightness <int>
```

Update a street light

```
update street-light <city_id>:<device_id> [enabled (true|false)] [brightness <int>]
```

Define a parking space

```
define parking-space <city_id>:<device_id> lat <float> long <float> enabled (true|false) rate
<int>
```

Update a parking space

```
update parking-space <city_id>:<device_id> [enabled (true|false)] [rate <int>]
```

Define a robot

```
define robot <city_id>:<device_id> lat <float> long <float> enabled (true|false) activity
<string>
```

Update a robot

```
update robot <city_id>:<device_id> [lat <float> long <float>] [enabled (true|false)] [activity
<string>]
```

Define a vehicle

Note that the commands to create a vehicle are consolidated instead of being split individually by vehicle type; to facilitate the polymorphism it has a mandatory type parameter

```
define vehicle <city_id>:<device_id> lat <float> long <float> enabled (true|false) type (bus|car)
activity <string> capacity <int> fee <int>
```

Update a vehicle

```
update vehicle <city_id>:<device_id> [lat <float> long <float>] [enabled (true|false)] [activity
<string>] [fee <int>]
```

Show the details of a device, if device id is omitted, show details for all devices registered with the city

```
show device <city_id>[:<device_id>]
```

Sensor commands:

Simulate a device sensor event; if a device id is omitted, simulate this events for all devices of this type registered with the city

```
create sensor-event <city_id>[:<device_id>] type (microphone|camera|thermometer|co2meter) value
<string> [subject <person_id>]
```

Send a device output:

Cause a broadcast to occur through the speaker of the specified device, or all devices in the city if the device identifier is omitted

```
create sensor-output <city_id>[:<device_id>] type (speaker) value <string>
```

Person Commands

Define a new Resident

```
define resident <person_id> name <name> bio-metric <string> phone <phone_number> role
(adult|child|administrator) lat <lat> long <long> account <account_address>
```

Update a Resident

```
update resident <person_id> [name <name>] [bio-metric <string>] [phone <phone_number>] [role
(adult|child|administrator)] [lat <lat> long <long>] [account <account_address>]
```

Define a new Visitor

```
define visitor <person_id> bio-metric <string> lat <lat> long <long>
```

Update a Visitor

```
update visitor <person_id> [bio-metric <string>] [lat <lat> long <long>]
```

Show the details of a person

```
show person <person_id>
```

Implementation details

In the natural hierarchy of devices it is reasonable to introduce additional levels: one is a subclass that will be a container of a blockchain account, to enable the devices inheriting from it to accept or disburse payment (since not all of them need this capacity), and another to be a parent class for all vehicle types (to contain attributes and methods common for all vehicles). The people hierarchy will only contain two levels. Similarly, there will be no additional structure in the hierarchy of sensors (but the parent class will exist to accommodate later design).

With regard to the APIs for defining and updating the objects, the “define” method of each pair will require an object of the correct type to be constructed and filled with data, and will then validate the attributes and insert it in the appropriate collection. The individual “update” methods will accept an identifier of an existing object and a collection of optional attributes; however, for convenience, polymorphic update methods for subclasses of Person and Device will also be available, accepting a template of the correct type where the non-null attributes of the template will be then set in the object. The text command client, even as it addresses the requirements to perform commands for each individual device and person type, will use these internally, allowing for more compact and maintainable code. This convention will require the atomic fields of the corresponding classes to be boxed by appropriate objects (Integer for int etc.); also, the text client internally will define and use factory template classes for creating instances of Device and Person objects

To allow for the requirement of the client that reads text commands and invokes the model service to execute them, the class representing the client will have an instance of the service and define a placeholder for the authorization token needed to communicate with the service. To address the separation of concerns, the task of converting text to commands and options will be split off into a parser class.

One addition to the explicit requirements is the presence of an “occupiedBy” field in the ParkingSpace object, representing the identifier of a vehicle currently parked there.

Exception handling

There are two classes of exceptions raised and handled by the design.

Frist is the ParserException, whose subclasses are thrown from the methods of the CommandParser class. The information in this exception notes the attempted operation and describes the specific reason why the parser could not process it. The only class that is aware of this class and, consequently, of this exception is the CommandProcessor, which utilizes the parser for converting the command strings into the model service API calls. All parser exceptions are caught there and wrapped in instances of ModelServiceException, with the error information transferred to them.

The ModelServiceException is raised whenever the API encounters a misconfigured input. The exception contains the reason for the error in its message. It is up to the client to handle the resulting situation. For example, the method of the CommandProcessor that reads commands from a file catches this exception and logs the message, and then continues processing.

Testing

The text-based command processor provides a convenient decoupled test framework. The test cases can therefore be implemented as a sequence of lines in a text file, and a simple text harness that accepts the file name as a command line argument and invokes the processor with it (and. optionally, uses the second argument as a name of file to which the output can be redirected) would allow to capture the results of executing this sequence. The harness is implemented as cscie97.smartcity.test.TestDriver, and two files with test cases have been provided

In the “smart_city_sample.txt” file the positive (functional) test cases are located. It exercises the creation and modification of all objects in scope, covering all commands described in the corresponding section and therefore fulfilling all functional requirements. No error messages are expected, and the details of the expected changes in objects’ attributes are provided in the comments. Since the range of possible commands and attributes is currently limited, this version of the implementation does not automate the verification of the output.

According to the testing-first design principle, the same test script was being used throughout the development process, with regression testing conducted after all significant changes to the design and implementation.

The results of running this test suite are included in the file “sample.out”.

The negative testing script verifies that the main categories of invalid input are treated correctly, with the appropriate exceptions raised and the state of the objects remaining consistent. It is provided in the file “smart_city_exception_handling.txt” and its results are in “exception.out”.

For performance testing, the test harness invoked without arguments measures times required to create 1 million of each of a Visitor and Car objects, and then invoking the API collecting the information on the city (which involves checking all Visitor locations against the coordinates of the city). The results are: 3.9 seconds for Car definition, 1.9 seconds for Visitor definition, and 2.3 seconds to verify locations.

Risks

As we are still in the intermediate phase of the design of the overall solution, the assumptions made concerning the objects and behaviors not yet specified (the event and output handling) may prove to be incorrect. This will require redesign and reimplementation of the corresponding sections of the software; however, as the effort has not been significant to date, and the components are reasonably encapsulated and decoupled, the risk is not too great.

The testing harness relies on human involvement to verify the results of functional testing (both positive and negative). As the scope of the project grows, this may turn out to be insufficient – both false positive and false negative may come up in regression testing. In this case, additional effort will need to be expended in automating the verification, but this is an addition to the existing functionality that will not require refactoring of the existing code. Similarly, when the current module becomes a part of the overall solution, new tests (integration) will need to be designed, and more attention will need to be paid to the performance testing as we scale the database to multiple cities and implement additional functionality (e. g. the above mentioned event and output handling).