CSCIE97 Software Design

House Mate Model Service Design Document

Date: 2017 October 3

Author: Jeremy Clark

Reviewer(s):

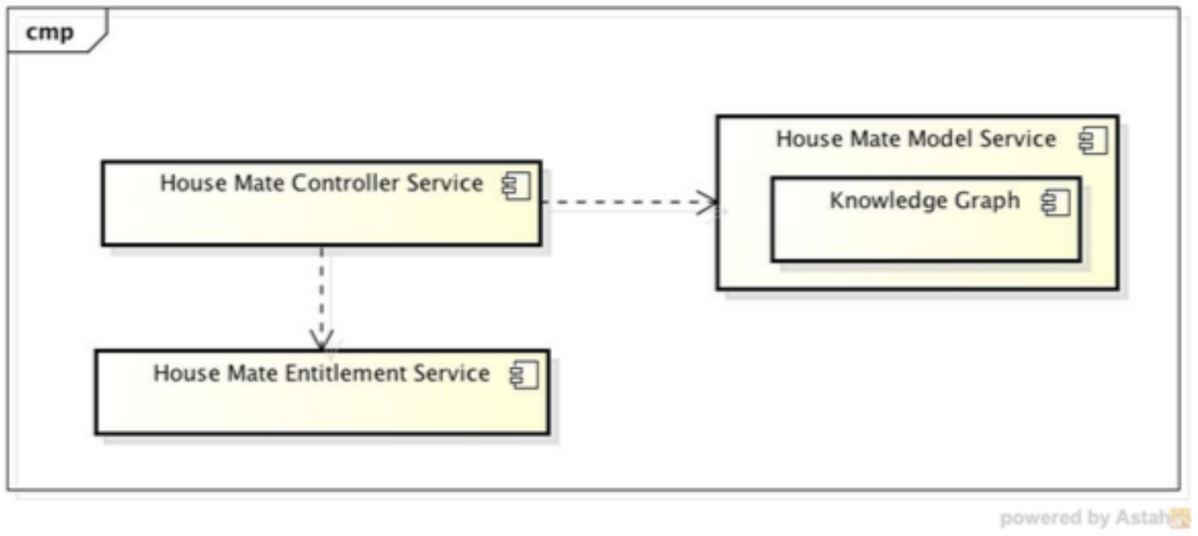
Introduction

This document defines the House Mate Model Service design.

Overview

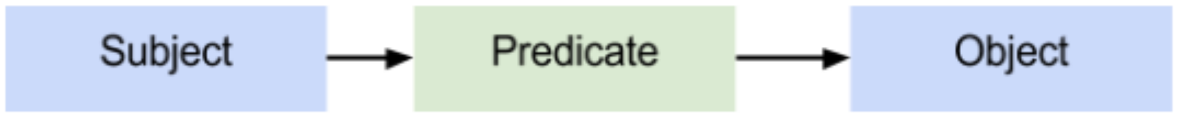
The House Mate home automation system is a comprehensive home automation solution; it supports multiple users with different roles in the home, allows for custom home layouts to support any living space and interfaces with myriad smart devices. The system is highly configurable, and is primarily controlled with voice commands from the home's occupants.

Here's a picture of the overall system architecture:

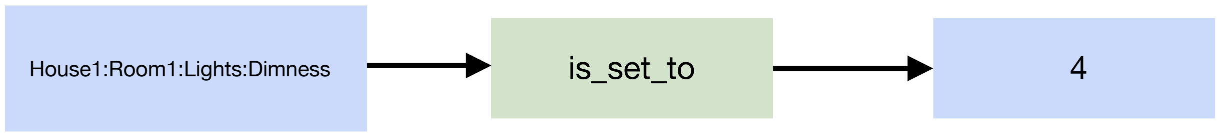


The subject of this document, the House Mate Model Service, is the system component that will manage the entire state of the "house", all of its rooms and occupants, as well as the state of all connected devices.

This information will be stored in a 'Knowledge Graph" which is made up of 'facts' about the home any any given time. The 'facts' are actually stored as "Triples" that represent the relationships between various entities in the House Mate Model. Triples look like this:



An example fact for the House Mate Model might be something like:



See the Knowledge Graph design document for details on the Knowledge Graph and Triples.

The House Mate Model Service will offer APIs to both query the state of the house as well as update the state of the house based on external events.

This following sections of this document will provide an overview of the component requirements, and outline the software design used to meet those requirements.

Requirements

In general, the requirement of the House Mate Model Service is to manage the state of the domain objects that underly the entire system. The system is made up of the following domain entities:

- House
- Room
- Occupant
- Devices
    - "Appliances", that have settings/commands

    - “Sensors” that measure/detect the environment
- Measures
- Settings

Specifically, the House Mate Model Service must provide a secure, programmatic interface (API), that is able to:

1) Support the creation of one or more **houses**;

2) Ensure that each **house** is unique within the entire system;

3) Allow **houses** to be attributed with a number of floors;

4) Support the creation one or more **rooms** that are part of a **house**;

5) Ensure that a **room** is only part of one **house,** and is assigned to a floor in that **house**;

6) Ensure that **rooms** are unique within a **house**;

7) Support the addition of **occupants** to the system, where **occupants** are either:

    1) **People**, which, in order to better support system-level permissions, must be further defined as either:

        1) Adults

        2) Children

    2) **Animals**; which are normally pets, but could be pests or intruders (bats, racoons, etc).

8) Allow associating an **occupant** to zero or more **houses**;

9) Allow disassociating an **occupant** from a **house**;

10) Support ‘anonymous’ or unknown **occupants** that are not associated with the house they are in (though they may be associated with other houses);

11) Support the addition of sensors**;** Which are **Devices** that provide one or more **Measures.**

12) Support the addition of appliances**;** Which are **Devices** that provide one or more **Settings.**

13) Ensure that **Devices** are globally unique in the system;

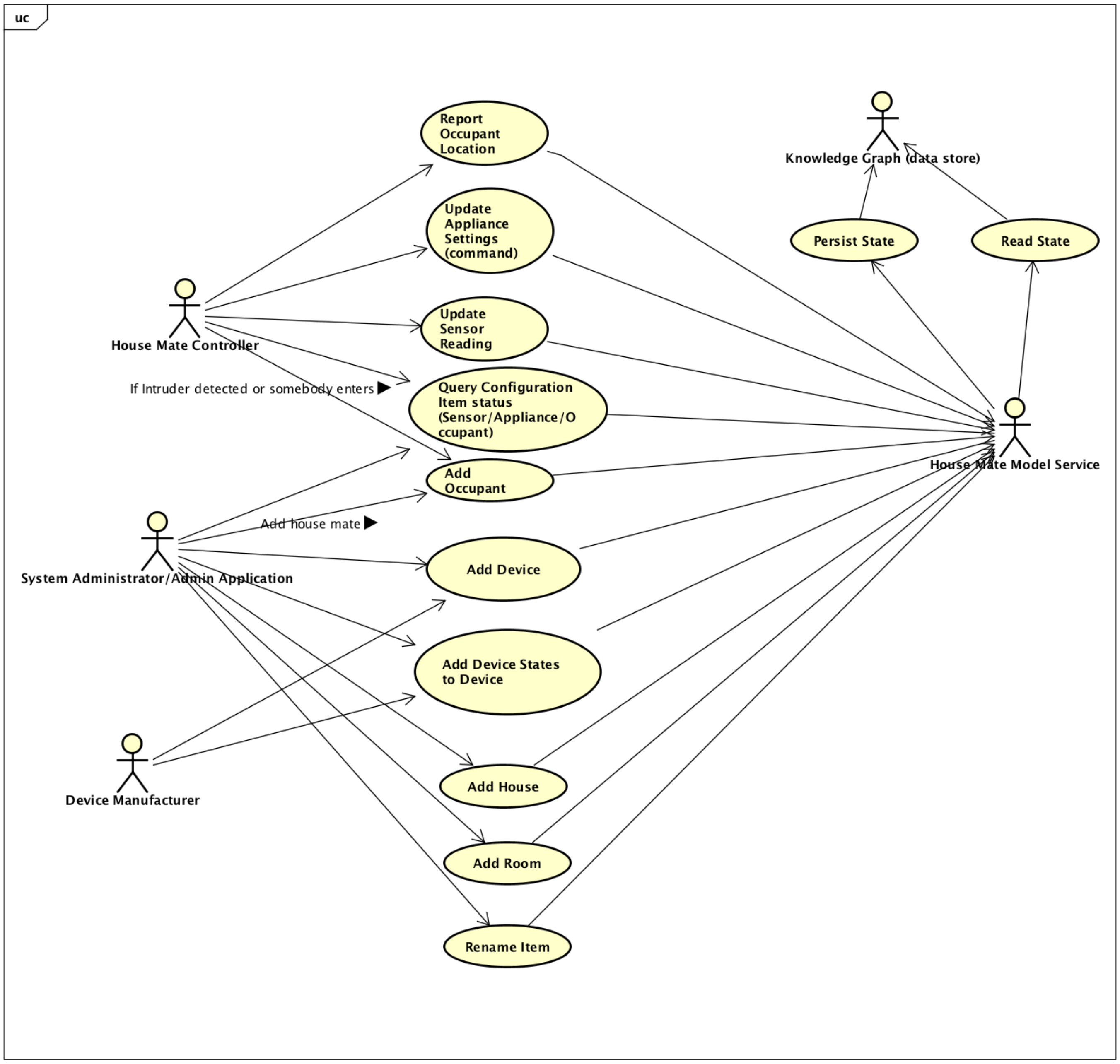
14) Attribute a **Device** by it’s location in a **room** (and by extension, its **house**);

15) Allow for changing any **Device State** of any **Device**

16) Support assigning a descriptive ‘display name’ for every **HMConfigItem** added to the system;

Use Cases

For use cases involving the House Mate Model Service, we will consider four actors:


uc
Knowledge Graph (data store)
Report Occupant Location
Update Appliance Settings (command)
Persist State
Read State
House Mate Controller
Update Sensor Reading
If Intruder detected or somebody enters
Query Configuration Item status (Sensor/Appliance/Occupant)
House Mate Model Service
Add Occupant
Add house mate
System Administrator/Admin Application
Add Device
Add Device States to Device
Device Manufacturer
Add House
Add Room
Rename Item

- The House Mate Controller
- The House Mate Model service
- A system administrator/adminstrative application
- A device/device manufacturer

Generally, as the House Mate Controller receives sensor input it queries or updates the House's state via House Mate Model service; then, based on the received input and the current state the Controller may send commands to the connected appliances in the house.

A system administrator can bootstrap the configuration of a house through an interface other than the House Mate Controller, like an administrative UI or CLI.

A Device could possibly update itself, by adding additional features (measures or settings) as part of a firmware upgrade or some other event.

Use Case Detail

Add House

Sys admin will add a house to the system, which can then be configured with additional commands.

Add Room

System administrator will add a room to a house configuration, in preparation for locating devices and occupants.

Add Device

Sys admin will add a device to a room. The device will be named and associated with a room.

Add Device State to System

Add a supported "Device State" to the system. For example, if a Device is meant to support reading temperature, a "Device State" will be registered in the model service with the name "temperature". The registered Device State will also define the valid value type; for the temperature example, this might be "Float". These device states can then be associated with Devices to provide functionality to the Device.

Add Device State to Device

Sys admin or firmware upgrade will add a supported “Device State” to a Device. A Device added through the “Add Device” use case can accept both “Settings” and “Measures” (the two types of device states). Adding a “Measure” to a device effectively makes it a ‘Sensor’ and adding a ‘Setting/ makes it an appliance. In this system, a device is a considered “Sensor” or an “Appliance” based on whether it has “Measures” or “Setting” device states, respectively. When a device state is added to a device, the Device State is added to a ‘pool’ of available device states within the Model Service itself, so they can be reused across devices in any House configuration.

Add Occupant

System administrator or House Mate Controller (via a sensor) adds an occupant to the Model Service (not associated with any particular house). The occupant must have a unique id (email address) and a name.

Add Occupant to To House

System administrator or House Mate Controller (via a sensor) associates an occupant to a house. The system administrator case would effectively create a ‘house mate’, ‘family member’ or ‘guest’ - essentially a known or expected member of the house. The House Mate Controller (via a sensor) would be ‘detecting’ an occupant in a house, and if not already associated, would make the association. For example, a stray cat, a human intruder, or even an occupant known to the system but not associated with the house. Say a neighbor with a similar system (so she is an occupant in the model service already) comes over unannounced, a camera/controller in the host’s house could make an ‘association’ to the host house.

Move Occupant

As an occupant moves about a house, a Controller can position an occupant with a specific room.

Update Measure

As a sensor (a Device that has Measures as part of its supported Device States) detects certain events in it’s environment, the “measurement” associated with the event can be persisted to the service.

Update Setting

An appliance (a Device that has Settings as part of its supported Device States), can have its “Settings” updated in order to update it’s control state. All features provided by an appliance will be expressed as “Settings”, and changing the state of a setting constitutes a “command”. For example, an “Oven” may have a ‘Target Temperature’ setting. If the “Target Temperature” setting is updated such that it’s new value is higher than its “Current Temperature” measurement - the oven will turn on its heat source until the target temperature is reached. The settings updates will usually come from the Controller via a sensor update or a voice command (which could be considered a special case of a sensor update). Settings updates can also be invoked by an Occupant manually adjusting a setting with a physical control on an Appliance.

Get Config Item State

A controller or sys aiming will query for the state of any item in the Model Service. This includes:

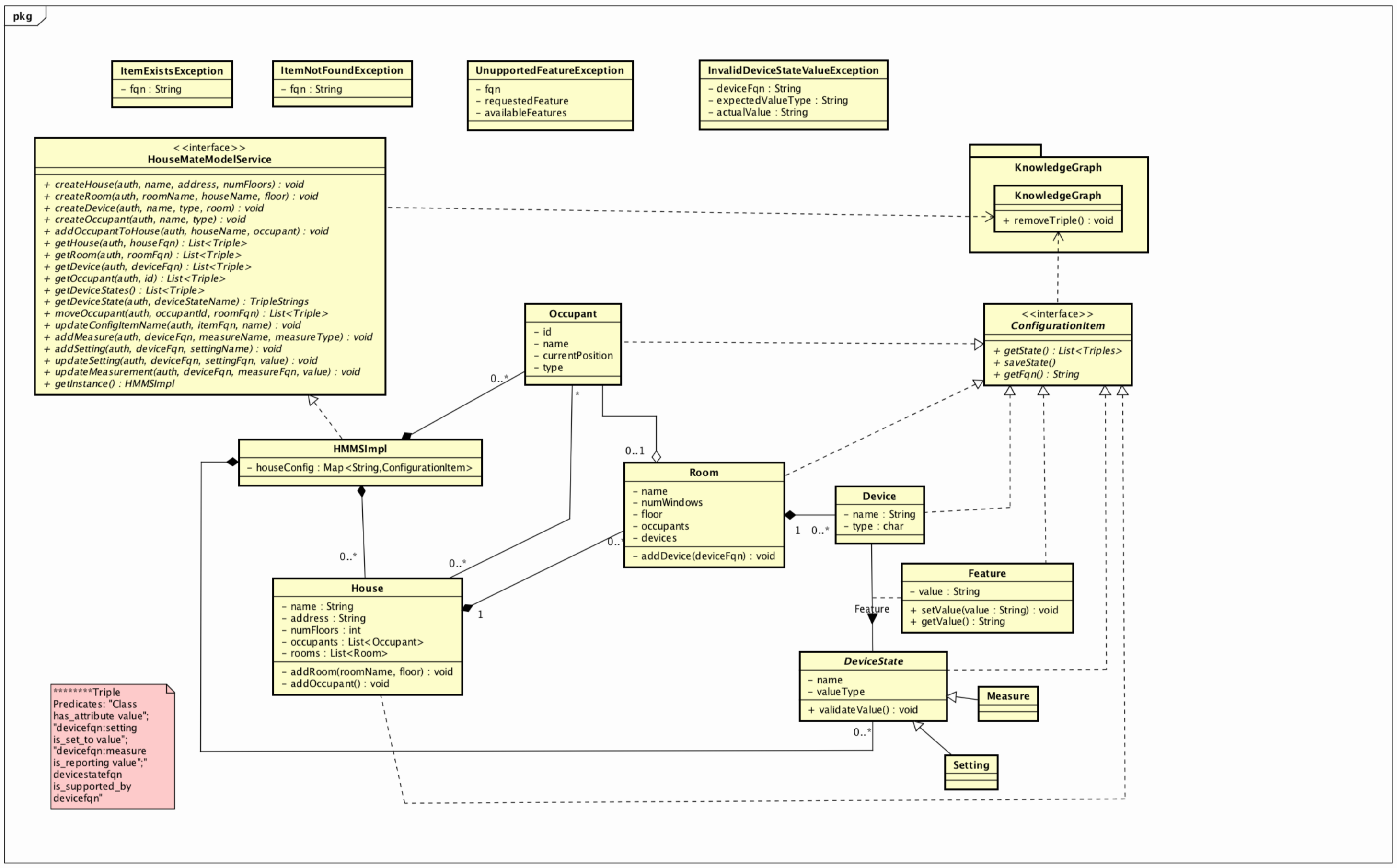
- Houses
- Rooms
- Devices
- Device States
- Occupants

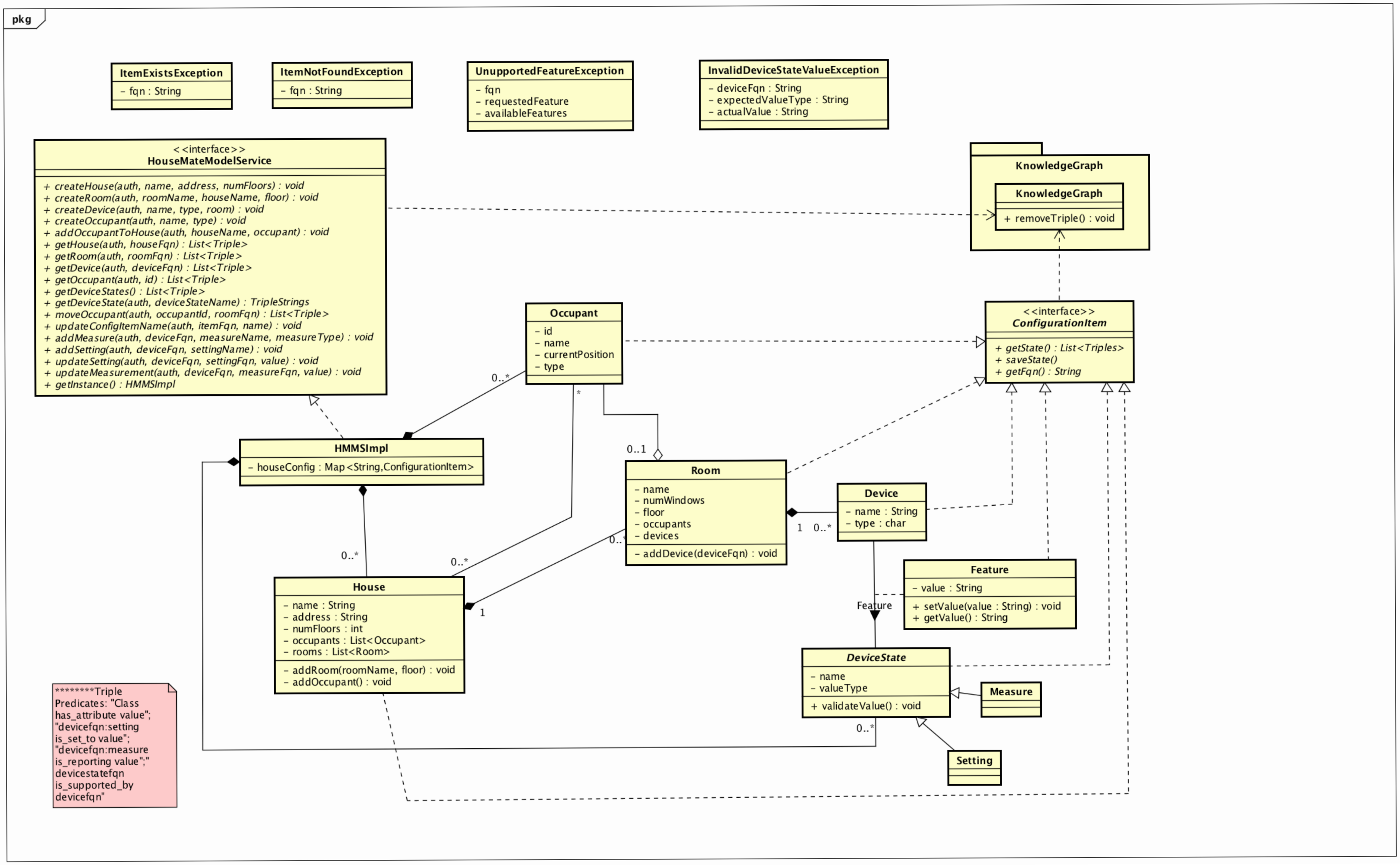
The config item state will be used to issue commands to appliances or to report status to an interested party.

Implementation

Once the Diagrams and Class Dictionary are reviewed for reference, please see the “Implementation Details” section for more explanations on the design.

Class Diagram


pkg
ItemExistsException
- fqn : String
ItemNotFoundException
- fqn : String
UnupportedFeatureException
- fqn
- requestedFeature
- availableFeatures
InvalidDeviceStateValueException
- deviceFqn : String
- expectedValueType : String
- actualValue : String
<<interface>>
HouseMateModelService
+ createHouse(auth, name, address, numFloors) : void
+ createRoom(auth, roomName, houseName, floor) : void
+ createDevice(auth, name, type, room) : void
+ createOccupant(auth, name, type) : void
+ addOccupantToHouse(auth, houseName, occupant) : void
+ getHouse(auth, houseFqn) : List<Triple>
+ getRoom(auth, roomFqn) : List<Triple>
+ getDevice(auth, deviceFqn) : List<Triple>
+ getOccupant(auth, id) : List<Triple>
+ getDeviceStates() : List<Triple>
+ getDeviceState(auth, deviceStateName) : TripleStrings
+ moveOccupant(auth, occupantId, roomFqn) : List<Triple>
+ updateConfigItemName(auth, itemFqn, name) : void
+ addMeasure(auth, deviceFqn, measureName, measureType) : void
+ addSetting(auth, deviceFqn, settingName) : void
+ updateSetting(auth, deviceFqn, settingFqn, value) : void
+ updateMeasurement(auth, deviceFqn, measureFqn, value) : void
+ getInstance() : HMMSImpl
KnowledgeGraph
KnowledgeGraph
+ removeTriple() : void
Occupant
- id
- name
- currentPosition
- type
<<interface>>
ConfigurationItem
+ getState() : List<Triples>
+ saveState()
+ getFqn() : String
0..*
*
0..1
HMMSImpl
- houseConfig : Map<String,ConfigurationItem>
Room
- name
- numWindows
- floor
- occupants
- devices
- addDevice(deviceFqn) : void
Device
- name : String
- type : char
1
0..*
0..*
0..*
0..*
House
- name : String
- address : String
- numFloors : int
- occupants : List<Occupant>
- rooms : List<Room>
- addRoom(roomName, floor) : void
- addOccupant() : void
1
Feature
- value : String
+ setValue(value : String) : void
+ getValue() : String
Feature
DeviceState
- name
- valueType
+ validateValue() : void
Measure
Setting
0..*
********Triple
Predicates: "Class
has_attribute value";
"devicefqn:setting
is_set_to value";
"devicefqn:measure
is_reporting value";"
devicestatefqn
is_supported_by
devicefqn"


pkg
ItemExistsException
- fqn : String
ItemNotFoundException
- fqn : String
UnupportedFeatureException
- fqn
- requestedFeature
- availableFeatures
InvalidDeviceStateValueException
- deviceFqn : String
- expectedValueType : String
- actualValue : String
<<interface>>
HouseMateModelService
+ createHouse(auth, name, address, numFloors) : void
+ createRoom(auth, roomName, houseName, floor) : void
+ createDevice(auth, name, type, room) : void
+ createOccupant(auth, name, type) : void
+ addOccupantToHouse(auth, houseName, occupant) : void
+ getHouse(auth, houseFqn) : List<Triple>
+ getRoom(auth, roomFqn) : List<Triple>
+ getDevice(auth, deviceFqn) : List<Triple>
+ getOccupant(auth, id) : List<Triple>
+ getDeviceStates() : List<Triple>
+ getDeviceState(auth, deviceStateName) : TripleStrings
+ moveOccupant(auth, occupantId, roomFqn) : List<Triple>
+ updateConfigItemName(auth, itemFqn, name) : void
+ addMeasure(auth, deviceFqn, measureName, measureType) : void
+ addSetting(auth, deviceFqn, settingName) : void
+ updateSetting(auth, deviceFqn, settingFqn, value) : void
+ updateMeasurement(auth, deviceFqn, measureFqn, value) : void
+ getInstance() : HMMSImpl
KnowledgeGraph
KnowledgeGraph
+ removeTriple() : void
Occupant
- id
- name
- currentPosition
- type
<<interface>>
ConfigurationItem
+ getState() : List<Triples>
+ saveState()
+ getFqn() : String
HMMSImpl
- houseConfig : Map<String,ConfigurationItem>
0..*
*
0..1
Room
- name
- numWindows
- floor
- occupants
- devices
- addDevice(deviceFqn) : void
Device
- name : String
- type : char
1
0..*
0..*
0..*
0..*
Feature
- value : String
+ setValue(value : String) : void
+ getValue() : String
Feature
House
- name : String
- address : String
- numFloors : int
- occupants : List<Occupant>
- rooms : List<Room>
- addRoom(roomName, floor) : void
- addOccupant() : void
1
DeviceState
- name
- valueType
+ validateValue() : void
Measure
0..*
Setting
********Triple
Predicates: "Class
has_attribute value";
"devicefqn:setting
is_set_to value";
"devicefqn:measure
is_reporting value";"
devicestatefqn
is_supported_by
devicefqn"

Class Dictionary

Interface::HouseMateModelService

Interface

Public API that allows access to the House Mate Model Service. All operations will require an auth token, and throw “UnauthorizedException” if an invalid token is passed for the requested API call.

Operations		
Operation	**Description**	**Throws**
public **createHouse**(auth, name, address, numFloors) : void	Adds a house to the system.	ItemExistsException ImportException UnauthorizedException
public **createRoom**(auth, roomName, houseName) : void	Adds a room to a specific house in the system Throws: ItemExistsException if similarly named room already exists in house. Throws: ItemNotFoundException house not found	ItemExistsException ItemNotFoundException ImportException UnauthorizedException
public **createDevice**(auth, name, type, room) : void	Adds a device to a room. Throws: ItemExistsException if similarly named device already exists in room. Throws: ItemNotFoundException if room not found	ItemExistsException ItemNotFoundException ImportException UnauthorizedException
public **createOccupant**(auth, uid, name, type) : void	Adds an occupant to the system. This does not associate the occupant with any specific house or room.	ItemExistsException ItemNotFoundException ImportException UnauthorizedException
public **addOccupantToHouse**(auth, houseName : char, occupant) : void	Associates an occupant with a specific house. An occupant can be part of 0 or multiple houses. Throws: ItemNotFoundException if house or occupant not found.	ItemNotFoundException ImportException UnauthorizedException

Operations		
Operation	**Description**	**Throws**

public **getHouse**(auth, houseFqn : char) : List<Strings>	Gets all the state for a specific house. It will be an outline of the following format. Each node in the outline is one or more Triple Strings:
	House FQN
	    House Attribute
	    Occupant ID (Occupants 'associated' with house)
	        Occupant Attributes and State
	    Room FQN
	        Room Attributes
	        Room occupants
	        Device FQN
	            Device Attributes
	            Device State FQN
	                Device State Attributes
	
	Throws: ItemNotFoundException if house doesn't exist.
		ItemNotFoundException
		QueryEngineException
		UnauthorizedException

public **getRoom**(auth, roomFqn : char) : List<Strings>	Will retrieve the description of a specific room, using a single outline node of the "Room FQN" type in the getHouse definition. Throws: ItemNotFoundException if roomFqn doesn't exist.	ItemNotFoundException QueryEngineException UnauthorizedException
public **getDevice**(auth, deviceFqn : char) : List<Strings>	Will retrieve the description of a specific device, using a single outline node of the "Device FQN" type in the getHouse definition. Throws: ItemNotFoundException if deviceFqn doesn't exist.	ItemNotFoundException QueryEngineException UnauthorizedException
public **getOccupant**(auth, id : char) : List<Strings>	Will retrieve the description of a specific occupant, using a single outline node of the "House FQN::Occupatnt FQN" type in the getHouse definition. Throws: ItemNotFoundException if occupant id doesn't exist.	ItemNotFoundException QueryEngineException UnauthorizedException

public **getDeviceStates**() : List<Strings>	Will retrieve the description of all device states, with the following for each device state:
	
	Device State FQN (Measure|SettingDevice State name
	      has_value_type {value_type}
	      is_supported_by {device_fqn}
		ItemNotFoundException
		QueryEngineException
		UnauthorizedException

Operations		
Operation	**Description**	**Throws**
public **getDeviceState**(auth, deviceStateName) : List<Strings>	Gets a single device state, as described in getDeviceStates(). *Throws: ItemNotFoundException if device state ID doesn’t exist.*	ItemNotFoundException QueryEngineException UnauthorizedException
public **moveOccupant**(auth, occupantId : char, roomFqn : char) : List<Strings>	Places an occupant in a room. Returns the same list of Triples describing the occupant state after the move. *Throws: ItemNotFoundException if occupant or room doesn't exist.*	ItemNotFoundException QueryEngineException UnauthorizedException

public **addMeasure**(auth, measureName, measureType) : void	Creates a 'Measure' (Device State) if it doesn't exist yet.
	addMeasure(auth_token, OvenTemperature, int)
	addMeasaure(auth_token, SurfaceTemperature, int)
		ItemExistsException
		ImportException
		UnauthorizedException

public **addSetting**(auth, settingName, settingType) : void	Creates a 'Setting' (Device State) if it doesn't exist yet For example: *addSetting*(auth_token, TargetTemperature, int) *addSetting*(auth_token, DoorLock, bool) *addSetting*(auth_token, burner_1_heat_level, “HIGH I MED I LOW”)	ItemExistsException ImportException UnauthorizedException
public **addFeature**(auth, deviceFqn, deviceStateFqn) : void	Adds the DeviceState (Setting or Measure) at deviceStateFqn to the Device at deviceFqn as a behavior. For example: **addFeature**(auth_token, “Setting:TargetTemperature”, “House1:Room1:Oven1”)	ItemNotFound ImportException UnauthorizedException

Operations		
Operation	**Description**	**Throws**
public **updateDeviceValue**(auth, deviceFqn, deviceStateFqn, value) : void	Set the value of the deviceStateFqn on deviceFqn to value **For example:** Updating a the target temperature on an Oven: **updateDeviceValue**(auth_token, “House1:Room1:Oven1” “Setting:TargetTemperature”, 350) Updating a the currently detected temperature of an Oven: **updateDeviceValue**(auth_token, “House1:Room1:Oven1” “Measure:OvenTemperature”, 228.4)	
public **getInstance**() : HMMSImpl	Gets singleton instance of HMMS.	

Associations		
Association	**Type**	**Description**
houses	List<Houses>	The houses that are available for configuration in the service
occupants	List<Occupants>	The list of occupants associated with the service
dataStore	KnowledgeGraph	The data storage layer for the state of every house and occupant in the system
deviceStates	List<DeviceStates>	The registered device states (measures and settings) that are available to all devices in the system.

Class::HouseMateModelServiceImpl

Implements HouseMateModelService

This is an insecure implementation that expects an auth parameter of “1” to authorize all API calls.

Other than implementing the public API, this class has the following:

Attributes		
Association	**Type**	**Description**
houseConfig	Map<String,ConfigurationItem>	Maintains a map of all ConfigurationItems in the system. All Houses, Rooms, Devices, Measures, Settings, Occupants. The keys are the fqn's, and the values are the insantiated domain objects. Adds items as they are created via the API creation methods. Reads from map whenever an API call has a dependency on an existing item, or to check whether a duplicate item is being created on a create method.

Interface::ConfigurationItem

Interface

Common Interface for items in the model. Defines methods that are used to identify items and store/retrieve state descriptions.

Realized By:

- House
- Room
- Device
- DeviceState
  - Measure
  - Setting
- Feature
- Occupant

Each realizing class implements a "getFqn()" method. An FQN (Fully Qualified Name) is a derived identifier that makes an ConfigurationItem unique within the system. See the individual class dictionaries for more details.

Operations		
Operation	**Description**	**Throws**
getState() : List<String>	Retrieves a line-delimited string of triples as a state representation of implementing entity. Implementors can add specific logic. See getHouse, getRoom, getDevice, getDeviceState, getOccupant in the HouseMateModelService Interface class dictionary.	QueryEngineException
saveState() : void	Saves the state of the ConfigurationItem to the KnowledgeGraph. Implmentors will add any entity specific logic for storing state.	ImportException QueryEngineException
getFqn() : String	Gets unique identifier for a House Mate Model configuration item. (Hosue/Room/Device/DeviceState/ Occupant). Fqns are colon-delimited strings whose form is dictated by the implementor.	

Class::House

Implements ConfigurationItem

Root node of a House Mate configuration. All Rooms, Devices, and Features within the service are related to a single house.

Attributes	
Attribute	**Description**
private **name** : String	Name of the house.
private **address** : String	Address of the house. Must be unique within the system.
private **numFloors** : int	Number of floors in the house

Associations	
Operation	**Description**
private **occupants** : List<Occupant>	Occupants associated with the house, i.e., household members.
private **rooms** : List<Room>	The rooms in the house

Operations	
Operation	**Description**
addRoom(roomName : String, floor : int) : void	Adds an room to the house
addOccupant(occupant: Occupant) : void	Associates an occupant to the house

Class::Room

Implements ConfigurationItem.

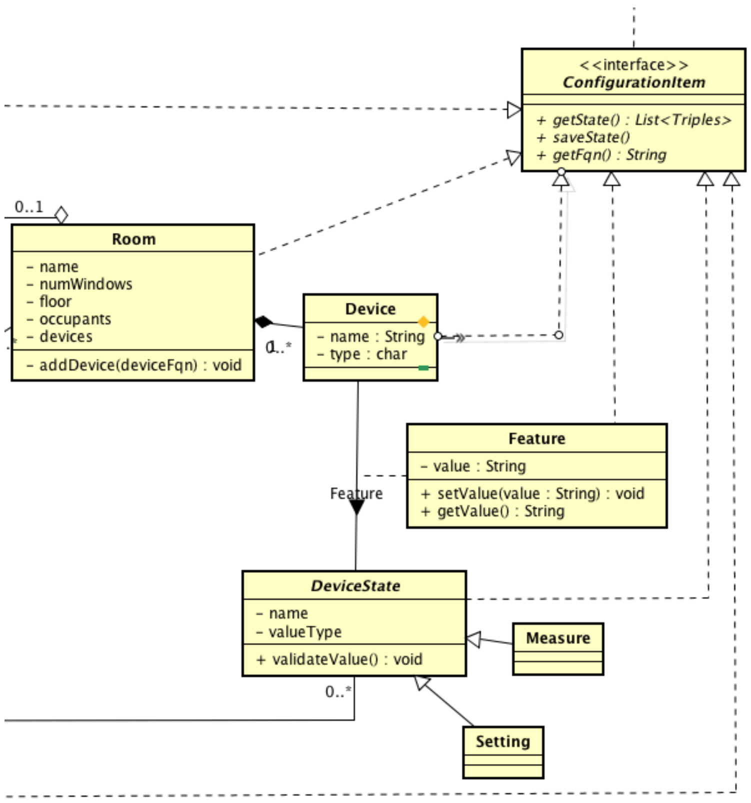
Room within a house configuration.

Attributes	
Attribute	**Description**
private **name** : String	The name of the room. Will be used in the fqn().
private **numWindows** : int	Number of windows in the room.
private **floor** : int	Floor of the house that contains the room.

Associations	
Association	**Description**
private **occupants** : List<Occupant>	
private **devices** : List<Device>	

Operations	
Operation	**Description**
protected **addDevice**(deviceFqn:String) : void	Adds a device to a room. Actually creates the device

Class::Device



Implements ConfigurationItem.

See Feature, Device State, Setting, Measure

Smart 'Device' within a home configuration. A device has a collection of "Features" which associate DeviceStates with the Device. DeviceStates have two subtypes: Setting and Measure. A device that supports "Settings" can be considered an Appliance and a device that supports "Measures" can be considered a Sensor. A device can support both Measures and Settings, in which case it might be called an Appliance, but technically it's just a Device that BOTH measures/detects things in its environment and is also takes commands to perform actions.

While there is no explicit Sensor and Appliance classes, this property could be derived by whether a Device has any "Settings" as Features.

Attributes	
Attribute	**Description**
private **name** : String	The name of the device, used in the FQN.
private **type** : char	The type of the device. This is meant to be a vendor-specific categorization, and doesn't map to an explicit type in the system. A devices 'type' is derived by the Features it has (Measures and Settings)

Associations	
Association	**Description**
private **features** : List<Feature>	

Operations	
Operation	**Description**
protected **addFeature**(deviceStateFqn:String) : void	Creates a Feature for the passed in DeviceState and adds it to the features list.

AbstractClass::DeviceState

Abstract, Implements ConfigurationItem.

See Feature, Device, Setting, Measure

Abstract class representing states that can be managed by Devices in the system. There are two concrete subclasses: Setting and Measure; Settings allow for devices to be controlled and Measures allow for Devices to collect facts about things they measure. DeviceStates are added as Features to devices. If a Device Features a Setting, it can be considered an appliance; Devices featuring Measures are Sensors. Devices with both can be considered appliances for all intents and purposes. DeviceStates are stored in the House Mate System being deployed, and can be reused to define Features across Devices in different houses.

Attributes	
Attribute	**Description**
private **name** : int	The device state name. Used in the deriviation of it's FQN.
private **valueType** : String	The supported type of the DeviceState. The valid values are “String”, “Integer”, “Float” or a “|”-delimited string, where each item is a valid “State”. The last is ideal for enum-like device state values, like “on|off|standby” The valueType will drive the logic in the “validateValue()” operation

Associations	
Association	**Description**
private **features** : List<Feature>	The features of which this DeviceState is a part.

Operations	
Operation	**Description**
protected **validateValue**(value:String) : boolean *Throws: InvalidDeviceStateValueException*	If the valueType is “String”, then any string value is a valid value for this device state. Same concept with “Integer” and “Float”. For the “|”-delimited strings, the only valid values are the delimited items in the string.

Class::Setting

Subclass of DeviceState
Implements ConfigurationItem

See Feature, Device, Measure

Represents controllable "State" of a Device - effectively making it a controllable Appliance.
The key difference from sibling subclass "Measure" is in the implementation of "saveState()" as it uses a different predicate to represent the facts in the KnowledgeBase.

Class::Measure

Subclass of DeviceState
Implements ConfigurationItem

See Feature, Device, Setting

Represents measured or detected "State" of a Device - effectively making it a Sensor. The key difference from sibling subclass "Setting" is in the implementation of "saveState()" as it uses a different predicate to represent the facts in the KnowledgeBase.

Class::Feature

AssociationClass for Devices <-> DeviceStates, Implements ConfigurationItem.

See Device, DeviceState, Setting, Measure

An association class representing the availability of a DeviceState on a Device. It holds the state 'value' of a DeviceState on that Device. For example, if a Device measures "temperature", a Device (probably named ‘thermometer’ but it can be named anything) has a Feature that relates a Measure object with name=“temperature” as its DeviceState. The current value of that feature would be the temperature of the thermometer.

Attributes	
Attribute	**Description**
private **value** : String	The current state value of a particular DeviceState on the supporting Device.

Associations	
Association	**Description**
private **deviceState** : DeviceState	The associated Measure or Setting
private **device** : Device	The associated Device

Operations	
Operation	**Description**
public **setValue**(value : String) : void Throws: InvalidDeviceStateValueException	Sets the value of the feature. Will call validateValue(value) of the associated deviceState to make sure that the passed in value matches the expected data type for that deviceState.
public **getValue**() : String	Returns the value of the feature as a String.

Class::Occupant

Implements ConfigurationItem.

An Occupant within the system. Occupants are of various types, which can be entitled to different interactions with the HouseMate System.

Attributes	
Attribute	**Description**
private **id** : char	The uniqueId of the occupant.
private **name** : string	The name of the occupant. Will be stored with spaces replaced with "_"
private **type** : String	The type of the Occupant

Associations	
Association	**Description**
private **currentPosition** : Room	The Room in which the occupant is currently located.
private **houses** : List<House>	The houses of which an occupant is a member.

Operations	
Operation	**Description**
protected **addHouse**(house : House) : void	Adds a house to the occupants set of houses.
public **setRoom**(room: Room) : void	Sets the current room of the Occupant

Exception Handling

In addition to ImportException and QueryEngineException from KnoweldgeGraph, the following additional exception classes are available. See class dictionary for what methods throw errors.

UnauthorizedException

Thrown when passing an invalid auth token to an API call in HouseMateModelService.

ItemExistsException

Thrown when trying to add an item that already exists.

Attributes	
Attribute	**Description**
private **fqn** : String	The fully qualified name of the item that tried to be added but already exists.

ItemNotFoundException

Thrown when operation depends on an item that doesn't exists in the configuration.

Attributes	
Attribute	**Description**
private **fqn** : String	The fully qualified name that was requested or depended on, but not found. For example, will be thrown when adding a room to a house that doesn't exist.

UnsupportedFeatureException

Thrown when trying to update a Device with a state that it doesn't support

Attributes	
Attribute	**Description**
private **deviceFqn** : String	The fully qualified name of the device
private **requestedFeature** : String	The name of the requested feature

Attributes	
Attribute	**Description**
private **availableFeatures** : String	The features enabled on the deviceFqn

InvalidDeviceStateValue

Thrown when trying to update a Device state with an invalid value for that device state. For example, “set Oven1 temperature UNLOCK”

Attributes	
Attribute	**Description**
private **deviceFqn** : String	The fully qualified name of the device
private **invalidValue** : String	The invalid value that was passed in
private **expectedValue** : String	A string reporting the expected value of the registered DeviceState on the request.

Implementation Details

Connections and Data

The entry point to the model service is the HouseMateModelImpl class. Since it's part of the housemate package, it has direct access to the model objects and exceptions. For each public API method, the actual domain objects will be instantiated and added to a 'full-index' of all ConfigurationItems, where the keys are the FQNs of the items.

When any method is called in the HMMS, it will use it's master index to see if the requested object actually exists or not, by looking up the FQN in the index. As such, when using the command interface, the target objects must be referred to by the FQN.

In general, when an creation/addition API method is called updates the state of one object. Each ConfigurationItem class is responsible for defining what it 'stores' about itself.

When calling a 'read' API method, the getState() method is called on the configurationItem, which is responsible for determining what triples to return.

Because each ConfigurationItem implementor implements it's own getState() and setState() method, they all import and have a dependency on the KnowledgeGraph.

In each ConfigurationItem class that has a collection of children, a list or map of those children should be kept on that class for referential purposes to help with reading and writing state. (For example, House maintains a list of Room and Occupant, Room maintains a list of Occupant and Device, etc.)

KnowledgeGraph

The KnowledgeGraph will need to be extended to support the removal of triples. In addition, the KG Importer class can take a list of Triples to remove.

Special notes on dynamic DeviceStates

My design takes the approach of 'dynamically defined' DeviceStates, with can then be associated with Devices to imbue them with behavior. Measures (subclass of DeviceState) imbue 'sensing' behavior, and Settings (subclass of DeviceState) imbue appliance behavior. I call the relationship between a Device and supported DeviceState a Feature.

As such, there will need to be “CreateSetting”, “CreateMeasure” and “AddFeature” on the public API.

Creating Settings and Measures

- CreateSetting will create a new setting, with a name and valid type.
- CreateMeasure will create a new Measure with a name and valid type.
- When defining a DeviceState, the user can establish the required type with is passed in as a string which represents a class name. The supported values are:
- Float, Integer, Boolean, String
- In addition, a “|” delimited string can be passed in, which will allow values that match any of the delimited-values. This is effectively the dynamic creation of an enum type.

Adding Features

- AddFeature will associate a Device and a DeviceState, and once done, that Device (sensor or appliance) will subsequently be able to

If a user tries to AddFeature to a device that doesn’t exists, an error should be thrown. If a user tries to AddFeature for a DeviceState that doesn’t exists, and exception should be thrown. If a user tries to Create a setting or measure that already exists, an exception should be thrown. If a user tries to set a DeviceState on a device that doesn’t have that feature, an exception should be thrown. If an invalid value is passed when setting a Feature value, and exception should be thrown.

Special note about Devices and Features

You’ll notice the ClassDiagram has no reference to “sensors” or “appliances”.

While the Command Interpreter can accept ‘create sensor|appliance’, ‘set appliance|sensor’ and ’show sensor|appliance’, the underlying class of the sensor/appliance is always Device - as such, these commands are complete interchangeable. The key part of this, is that the “State” being set in the ‘set sensor|appliance’ must be a fully qualified DeviceState, which includes whether it’s a measure or a setting…for example:

As long as the Oven has the Setting:target_temperature feature, these two commands would do the same thing:

set sensor House:Kitchen:Oven state **Setting:target_temperature** value 375

set appliance House:Kitchen:Oven state **Setting:target_temperature** value 375

This is a little strange, but I add both nouns (sensor and appliance) to meet the provided requirements. Ideally, I would implement a command like “set device House:Kitchen:Oven state Setting:target_temperature value 375”, and the fact that this is an ‘appliance’ is derived by the fact that we’re updating a ‘Setting’. Likewise, the command “set device House:Kitchen:Oven state Measure:oven_temperature value 273.7” would imply that the oven device is also a “sensor”. Any device can be an applicance, sensor or both.

Updating a Setting would be akin to ‘sending a command’ as it will change the configurable state of a Device (appliance). If the entitlement-based control is one of the goals of separating appliances from sensors, then those entitlements could be applied more granularly to whether the requested update is of a Measure (i.e. sensor) or a Setting (i.e. appliance).

Testing

Implement a test driver class called TestDriver that implements a static main() method. The main() method accepts a script file that has command lines with the format listed in the requirements document. In addition, my design specifies dynamic creation of device states, and as such supports commands of the following form:

“define device {setting|measure} {name} type {type}”, where type is one of “String”, “Integer”, “Float”, or a “|” delimited string that represents a state enum, like “On | Off | Standby”.

“add feature {settingFqn | measureFqn} to {deviceFqn}”

These commands will be used at the beginning of the scripts to bootstrap the system with states and to associate states with devices

Risks

- **My Device/DeviceState/Feature** model (rather than explicit Sensor and Appliance classes, with pre-defined types) definitely puts an emphasis on flexibility and run-time configuration. While this aims to meet the needs of a potentially large and changing topography of devices, there are clear risks:
  - Dynamically defining acceptable types via a command has inherent risks, but they could be mitigated with further requirements clarification and code hardening. Especially when trying to allow for the creation of dynamic Enum types.
  - Device behaviors (DeviceStates) must be added as part of the configuration of the system.
  - Any behavioral logic requiring a difference between Appliance/Sensor will have to use derived logic about the Features that a Device supports rather than a Compile-time type alignment.
  - It might reflect this anti-pattern: https://en.wikipedia.org/wiki/Inner-platform_effect
- **My lack of explicit types for Occupants** should probably be enhanced, although I wasn't sure on how to best achieve this.
- **Every implementor of ConfigurationItem depends on KnowledgeGraph** and accesses it directly. It seems like it might be cleaner to consolidate all data I/O to a single class.
- **KnowledgeGraph I/O operations are atomic.** That is, when state is updated for a configuration item ALL current state for that item is removed and the new state is written.