House Mate Controller Service Design Document

Date: October 7, 2015

Author: Kush Patel

Reviewer(s): Sarah Leinicke, Joshua Mcelfresh

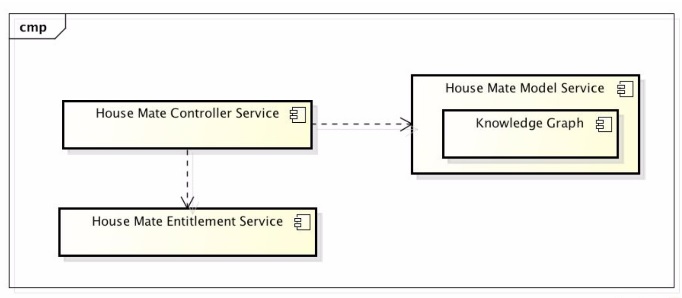
# Introduction

This document describes a design and implementation of the House Mate Controller Service (HMCS); a controller service that works in conjunction to the House Mate Model Service (HMMS). Together these systems control and maintain the state of many houses and occupants assisting them in day to day household activities.

Overview

The HMMS is a singleton that merely maintains the state of all the occupants, houses, rooms, and devices. It’s a great way to check the state of the system, but not control it. This is where the HMCS comes into play; a system that can interact with different states of the occupants and houses as well as different commands provided to it. Automating things such as turning off the lights as you leave a room or fall asleep or alerting the user that the food in the oven has been cooked. Users can also interact with it by passing commands. For example, a user can say “open window” and the HMCS will determine the room the occupant is in, and open the windows to that room.

The following design shows how the HMCS fits in with the rest of the system.



This is a very high level design overview of the entire House Mate system. The HMCS implements a command pattern with which it interacts with the HMMS (shown below in detailed class diagram).

# Requirements

The HMCS is responsible for monitoring the sensors and appliances within the home. Upon changes in the state of the sensors or appliances, the HMCS must execute the proper commands that complete the system interaction for the respective state change. For example: if the HMCS is given the command to “open window” the HMCS must notify the HMMS and change the state of the window to open. Specifically, the HMCS must support the following functions:

* Monitor Sensor and Appliances for state changes
  + Respond to the state updates in sensors and appliances and apply rules to generate actions
* Accept voice commands via the Ava appliances and provide voice feedback when necessary
  + Act on given voice commands and control appliances accordingly.

# Use Cases

Occupants interact with the HMCS via input appliances, Ava, to control other appliances around the house. Below is a sample flow of the interaction between the actors and the system.

# 

Use Case: Occupant/s

1. Occupant says command “open windows”
2. HMCS receives command and calls Control Window
3. Control Window opens window and passes state change to HMMS notifying the windows are open.

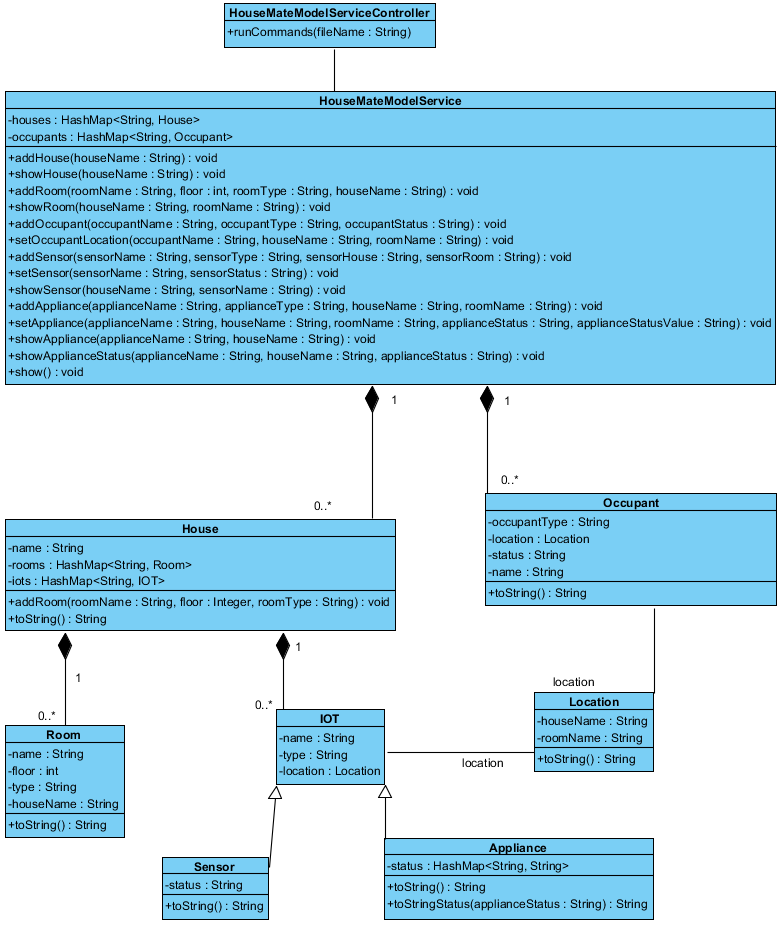
Use Case: Sensor

1. Smoke Detector senses “FIRE” and sends information to HMCS
2. HMCS calls Control Lights
3. Control Lights turns on all Lights and notifies the HMMS for state change
4. HMCS sends voice response via Ava to occupants notifying them about the fire
5. HMCS dials 9-1-1 and contacts emergency services

# Implementation

# Class Diagram

The following class diagram defines the House Mate Model Service implantation and all classes used for this solution and their affiliations to the overall system.



# Class Dictionary

*\*\*NOTE: All methods are public.*

**HouseMateModelServiceController**

The HMMSController is responsible for reading each individual command and parsing them; then calling the appropriate method in the HMMS to either input the configuration, or print out information. The key to the runCommand feature is the format of the input; given that these would be controlled commands via an API, the format for each line should always be consitant; thus the command parser does error out due to poor input format.

**Methods**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| runCommands | (fileName : String) | Reads in all the lines from the filename provided; ignores lines with a “#” as they can be comments for a user reading the command list. Then runs each command step by step, so “show” commands should be at the end once all the configuration has been loaded. |

**HouseMateModelService**

The HMMS is a singleton class that gets initialized once the application starts which is important since its storing all the data and configuration for the whole House Mate system and so data isn’t accidentally set in a different new HMMS.

**Methods**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| addHouse | (houseName : String) : void | Method to add a new House to the system. Must have a unique houseName. |
| showHouse | (houseName : String) : void | Method that prints out the configuration of the housename provided, or HouseNotFoundException |
| addRoom | (roomName : String, floor : int, roomType : String, houseName : String) : void | Method to add a new room to a house; must provide a valid house, and a unique room name, integer floor value, and string type. Throws HouseNotFoundException if house is invalid. |
| showRoom | (houseName : String, roomName : String) : void | Method that prints out the configuration of the room. Throws HouseNotFoundException or RoomNotFoundException accordingly if the house name / room name are invalid. |
| addOccupant | (occupantName : String, occupantType : String, occupantStatus : String) : void | Method that adds an occupant into the HMMS. Must have unique name, a type, and status. |
| setOccupantLocation | (occupantName : String, houseName : String, roomName : String) : void | Method that gives an occupant a location. Must have a valid house name and room name, else it throws a HouseNotFoundException and RoomNotFoundException accordingly. |
| addSensor | (sensorName : String, sensorType : String, sensorHouse : String, sensorRoom : String) : void | Method to add a new sensor to the system. Must have a unique sensor name, a type, a valid house name and valid room name; else it throws a HouseNotFoundException and RoomNotFoundException accordingly. |
| setSensor | (sensorName : String, houseName : String, sensorStatus : String) : void | Method to set the status of a sensor, must have a valid house name and sensor name, else it throws a HouseNotFoundException and SensorNotFoundException accordingly. |
| showSensor | (houseName : String, sensorName : String) : void | Method to print out sensor information. Must have valid house name and sensor name, else it throws a HouseNotFoundException and SensorNotFoundException accordingly.  Returns location and status of the sensor. |
| addAppliance | (applianceName : String, applianceType : String, houseName : String, roomName : String) : void | Method to add an appliance to the HMMS, must have a valid house name, room name, and unique appliance name; else it throws a HouseNotFoundException and RoomNotFoundException accordingly. |
| setAppliance | (applianceName : String, houseName : String, roomName : String, applianceStatus : String, applianceStatusValue : String) : void | Configures an appliance: either update existing values or adding new status such as: temperature =400. Or add new one: ovenclean=needed |
| showAppliance | (applianceName : String, houseName : String) : void | Returns the appliance and all the status and values for it. |
| showApplianceStatus | (applianceName : String, houseName : String, applianceStatus : String) : void | Returns the specific value for the status request on the specific appliance. |
| show | (): void | Shows the configuration for all the houses and everything in the system |

**Properties**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| houses | HashMap<String, House> | Map of all the houses registered and configured in the system |
| occupants | HashMap<String, Occupant> | Map of all the occupants in the system, don’t necessarily have to be affiliated with a house, but are registered and configured. |

**House**

The house class is used to model a house that is configured and controlled by the HMMS.

**Methods**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| addRoom | (roomName : String, floor : Integer, roomType : String) : void | Method to add rooms to a house. Since each room is specific to a house, and can’t move from house to house. |
| toString | ():String | Custom toString to print information when “show” is called |

**Properties**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| Name | String | Unique name of the house |
| Rooms | HashMap<String, Room> | Map containing all the rooms in a house; each with a unique name to distinguish between similar rooms |
| IOTs | HashMap<String, IoT> | Map containing all the IoTs in a house; each with a unique name to distinguish between similar devices |

**Associations**

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| Houses | HashMap<String, House> | HMMS has many houses in its system each with a unique name to identify that house and then all the configurations affiliated for that house. |

**Occupant**

Occupant is any person / animal that is registered in the HMMS system. They do not necessarily have to be residents of the home, but can be anyone whose voice and facial recognition is in the system. This way the system can track all occupants in and out of a house, and identify any suspicious characters.

**Methods**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| toString | ():String | Custom toString to print information when “show” is called |

**Properties**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| Name | String | The unique name of the occupant |
| OccupantType | String | Adult, child, animal |
| Location | String | Not required, but gets set as the occupant enters a house and moves around the house, their location gets updated. |
| Status | String | Active / Sleeping (could be many more statuses such as “injured”) thus tracking specific states of a person as well |

**Associations**

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| Occupants | HashMap<String, Occupant> | Many people can be registered in the HMMS, whether they are residents of the house or friends or neighbors or anyone. This way the system can track all users it recognizes. |

**Room**

The room class is used to model a room in the house. It is used part of location to identify where occupants and/or IoTs are located within a house.

**Methods**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| toString | ():String | Custom toString to print information when “show” is called |

**Properties**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| name | String | Unique identifier of the room. |
| floor | Integer | The level of which this room is located in the house. |
| type | String | The kind of room, help user to identify bathroom at a neighbor’s house maybe. |
| houseName | String | The house in which this room is located. |

**Associations**

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| Rooms | HashMap<String, Room> | A house has many unique rooms. Possible to have multiple bedrooms, but they are still unique. |

**IOT**

The generic class of al interactive devices in a house that can provide information about the house, or complete actions such as: starting laundry, prepping oven, launch Roomba to vacuum.

**Properties**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| name | String | The unique name of the device |
| type | String | The type of the device |
| location | Location | The location(house + room) of the device |

**Associations**

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| House | HashMap<String, IOT> | A house a set list of devices, though there might be multiple similar devices like many smoke detectors, they all must have a unique identifier to say which one has error. |

**Location**

Not mentioned as part of the original design, but definitely a handy tool if generic locations want to be used in the future. Comprised of a houseName and roomName and so if a user may want to turn on the lights, but doesn’t know which lights are in this room, they could potentially use their location and match with all lights with the same “location” and turn them on.

**Methods**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| toString | ():String | Custom toString to print information when “show” is called |

**Properties**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| houseName | String | The house part of the location (since an occupant could be your neighbor looking to turn on your lights.) |
| roomName | String | The room in the house. |

**Associations**

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| IoT | Location | IoTs are all installed in specific locations |
| Occupant | Location | Occupants have a specific location they are |

**Sensor**

Sensors are a subset of IoT devices that capture and share data about the house. Each sensor has a name, type, location, and status. Example: a smoke\_alarm could have status: OK or FIRE or Battery\_Low. Whatever the status, communicating with the HMMS, the show sensor command will give you the status.

**Methods**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| toString | ():String | Custom toString to print information when “show” is called |

**Properties**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| status | String | The state in which the sensor is in. |

**Associations**

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| IOT | Subclass | IOT are the overarching devices that the house mate system controls / communicates with. Sensors are a subset or (subclass) and so they are an extension on it. |

**Appliance**

Appliances are also a subset of IoT devices but these are about to be controlled. These can have more elaborate states and values for their states. Example: if the oven is on, it would have a status: temperature with a value: 350.

**\*\*NOTE:** control/status for appliances was grouped to 1 hashmap property due to the similarity in their nature for an appliance. On/Off or 350/425 or Open/close are essentially state values for something that can be controlled, but in the end, simply only need a value to the key since we not defining many advance characteristics in this higher level, 1st iteration of the project.

**Methods**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| toString | ():String | Custom toString to print information when “show” is called |

**Properties**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| status | HashMap<String, String> | The map that stores a status and the value for the status. |

**Associations**

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
|  | Subclass | IOT are the overarching devices that the house mate system controls / communicates with. Sensors are a subset or (subclass) and so they are an extension on it. |

# Implementation Details

*Explain details of the implementation.*

This project requires the implementation of a service used to configure and manage an instance of a House Mate. A system that automates daily house activities as well as providing real time data / status updates when needed. The HMMSController is the command parser; any errors with the commands or the files are handled here; otherwise it calls the appropriate method in the HMMS to complete the action required for the command. The HMMS is a single entity, aka singleton that stores all the data for this current House Mate and prints out any information that has been requested. Once the application is initialized, the singleton is initialized and keeps that static state for the remainder of the application. The HMMS supports a Command Line Interface for configuring the houses, rooms, sensors, appliances, and occupants, as well as printing data / configurations for each of them. Essentially each object is just a data storage item, and the HMMS configures the correlation between them, and then stores the highest level objects (house and occupants) in itself.

# Testing

**Functional Testing:** providing a proper main method to call the HMMS system, and providing a proper input file. Run that file and verify that the output of the system is valid. Thus proving that the input/update commands ran successfully resulting in a properly configured HMMS.

**Performance Testing:** Cloning the input commands many times with different names to meet the unique requirements. Then running that file and monitoring the speed vs the small original input. Hardly any difference noticed, but this is also a small-medium scale application.

**Regression Testing:** Modifying the input file commands to have invalid names or inputs and verifying that the correct exceptions are thrown.

**Exception Handling:** A part of regression handling, all exception scenarios is tested. Also, instead of returning a massive stack trace, exceptions simply print out a human friendly error with the invalid input or filename for the user to investigate the input.

# Risks

This whole HMMS system is stored live in memory as the application runs, so if the command file in extremely large, building out a very large complex HMMS system, could run out of memory allocated to the JVM. The design assumes for small-medium files. Anything larger, then a proper database system to store the data would be recommended.