HouseMate Controller Service Design Document

Date: 10/28/18
Author: Brandon T. Wood
Reviewers: Gerald, Quanti

Introduction

This document defines the design and implementation plans for the HouseMate Controller Service. This service monitors the activity of sensors and appliances tracked by the HouseMate Service. It looks for changes that meet certain preset conditions and triggers actions based on them. An example would be a smoke alarm detecting smoke in a kitchen triggering a warning siren from the HouseMate speaker system. Below is a broader set of requirements, explanations and implementation plans for creating such a service.

Overview

This HouseMate Controller Service acts as a controller and invoker for the other HouseMate Services. It has a large and extensible list of possible update states that correlate to specific command messages. Updates are triggered by changes to the the model by sensors and appliances. The latter can be reconfigured by the HouseMate Service in response to events, such as a sensors noticing motion triggering a garage door to open. Every time an update triggers a command message it is neatly logged for later auditing.

Requirements

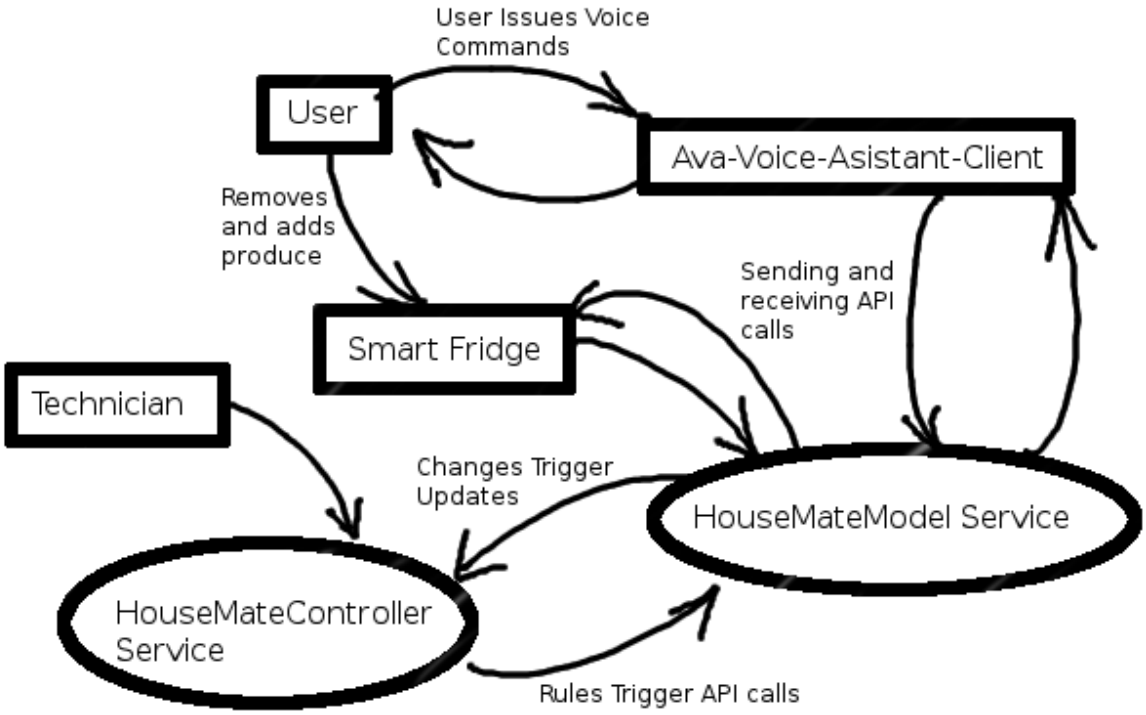
This service requires the use of both the Command Pattern and the Observer Pattern for implementation of communication between the Controller and Model. It also must make use of the previous work on the Knowledge Graph and HouseMate Model Service for tracking user whereabouts and sensors status. As usual this application should be performant, scalable and extensible for future modifications. It should be made robust and be well documented.

It must also support a series of predetermined voice commands such as a user commanding that lights or appliances change state and rules such as a rule that a beer count of less than 4 in a fridge signals an alert to order more.

Use Cases

The Controller Service must be extensible so that new appliances, sensors and rules can easily be integrated. For enthusiasts and vendors who want to add their own packages of actions we make the process of extending our existing ecosystem easy and well documented. Below we see someone using a voice command to the built in voice assistant, Ava, to trigger lights to be turned on. Additionally we see motion trackers detecting motion while our user is on vacation and sounding an alarm.

Below we show a user issuing voice commands to Ava the voice assistant and modifying the contents of the Smart Fridge. These events modify the underlying HouseMateModel and trigger preset behaviors from the HouseMateController. These changes could be measuring the inventory of the fridge or following commands from the voice assistant. Additionally technicians support the system can make use of Controller logging in order to make better troubleshooting decision when automation mishaps occur.



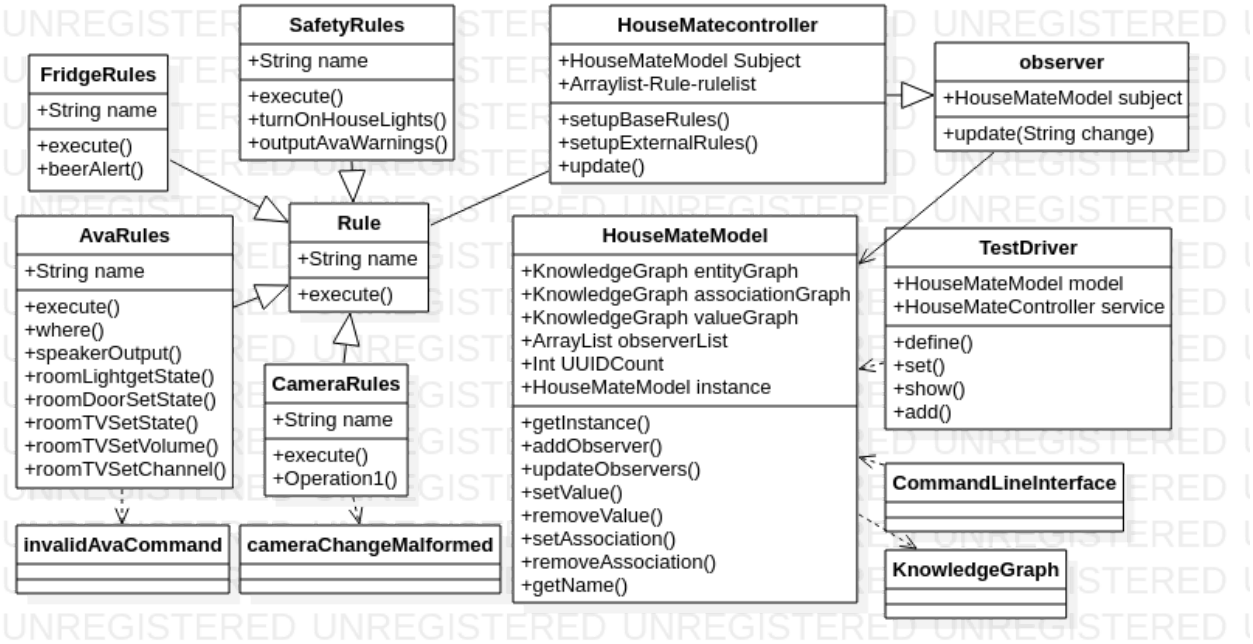
Implementation

Currently the implementation plan makes use of the Command Pattern to dispatch changes to the model API based on a set of rule classes and also the Observer Pattern to trigger this rule dispatching and parsing when changes are made to the model. Since we are leveraging the existing

HouseMateModel from our previous assignment most of the work is already done for that and the CLI input. Modifications will have to be done so that when values are changed in the valueGraph or assocationGraph it will alert the HouseMateController service with those changes.

Due to the model being used by both the controller and the CLI it may be best to also switch it over to a singleton model. Once the HouseMateController is alerted with those changes it will iterate over the rules, passing them the changes for parsing and actions. Then logging those that were triggered. Rules are also passed a reference to the model on which they make API calls based on their internal logic. Additionally occupant whereabouts will be stored in the associationGraph and maintained by cameras.

Class Diagram



Class Dictionary

HouseMate Model Service

For in-depth details about the model service please see it's design document. This model is a near real time representation of the smart homes under the HouseMate software's management. It acts as a datasource, transport layer and now an event publisher. Changes included the addition of classes to fulfill this duty and extend the API to allow further configuration by the HouseMate controller service. The new classes are documented below:

Method Name	Signature	Description
addObserver	void	Takes an observer object and adds it to the observer list.
updateObserver	void	Iterates over the list of observers and hands them the new changes given to this function as a String delimited by colons.
removeValues	void	Removes a value from the valueGraph, needs the three portions of it's triple.

Property Name	Type	Description
observerList	ArrayList<HouseMateController>	Holds the list of current observers that need to be updated when changes occur.

HouseMate Controller Service

The controller service implements form a super observer class and has two main functions, firstly it subscribes to an event publishing HouseMate model. It listens for changes to the subject by implementing the observer pattern. It consumes a model service reference at construction, then calls the addObserver method and passing a reference to itself. Now when changes are made to the models values the model calls the controller's update method. From that method the the controller service takes on its second function which is to look for certain types of changes and trigger events. (Thankfully based on my previous design it should be easy to localize changes to the setValue method of the model.)

It's second function is to be implemented as a command pattern, it will take many rule classes into a list and when update is called it will iterate over them, then pass them the needed information and references. The rules will be in charge of whether the changes are meaningful to them enough. Each rule set class encapsulates a set of rules that fall under a certain subject. As an example there would be a fridgeRule class that encapsulates any rules on dietary alerts, food inventory or fridge maintenance. It could also be assumed that anyone could easily add additional classes easily via this method.

Method Name	Signature	Description
setupBaseRules	void	Adds internal rules to the ruleList that the system needs to exist, uses Lambda functions.

setupExternalRules	void	Adds external rules found in classes, rules can be easily added and extended here.
update	void	Takes in changes from the subject it is subscribed. Iterates over rules in rule list and call their execute functions.

Property Name	**Type**	**Description**
subject	HouseMateModel	The reference to the model the controller is watching and executing on.
ruleList	Arraylist<Rule>	List of all rules that need to be checked on any change to the system.

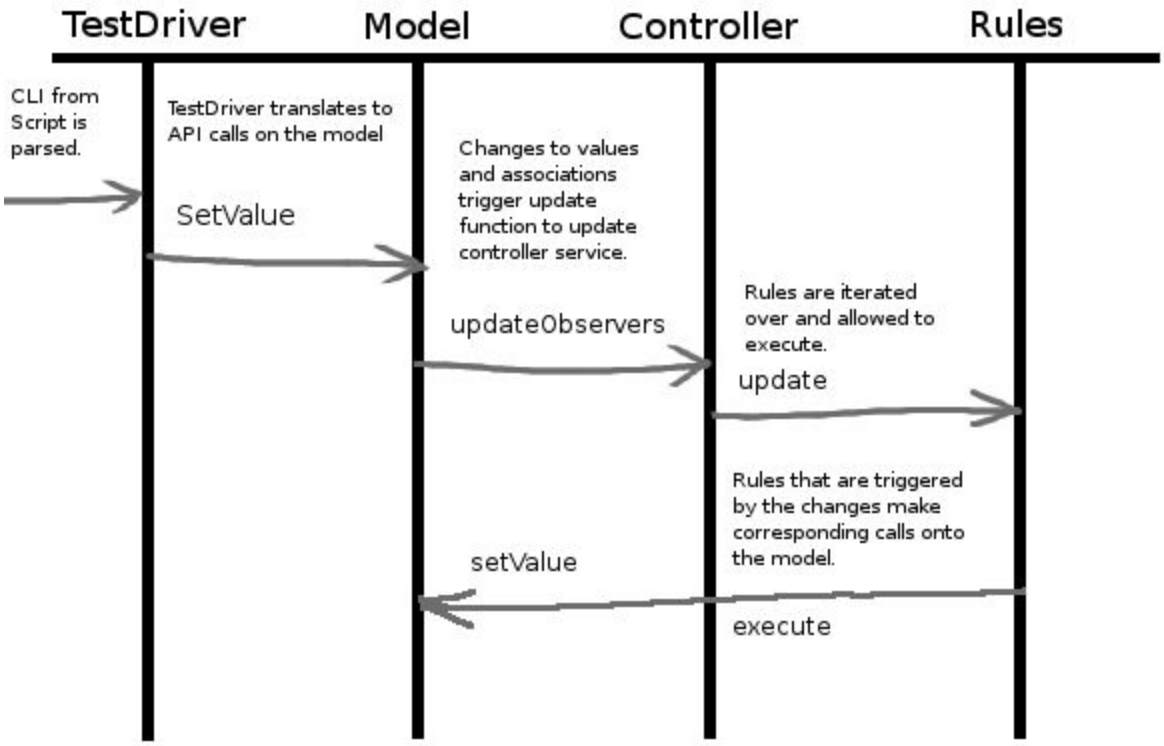
Rule Classes

For managing external rules there is a base Rule interface class that all rule sets implement from. They all make use of an execute method that is run and passed a change string and model in order to perform checks and actions respectively. The change string contains the value changed and the entity it is associated with. Rules extend this class by adding their own parsing triggers and logic to modify the model by.

Method Name	**Signature**	**Description**
execute	void	Adds internal rules to the ruleList that the system needs to exist, uses Lambda functions.

Property Name	**Type**	**Description**
subject	HouseMateModel	The reference to the model the controller is watching and executing on.
ruleList	Arraylist<Rule>	List of all rules that need to be checked on any change to the system.

Sequence Diagram



Above we see a generic interaction of all classes following a change to the valueGraph. A more literal example might be someone inputting a voice command to Ava. This comes in via the CLI and is parsed into our graphs using setValue. A change to setValue causes the model to call updateObservers which iterates over the list of observers, calling their updates and passing them the change in String format.

Next the update function of our controller iterates over it's ruleList and calls execute on all the rules and passing them the change and a a reference to the model. The rule decides based on the change if it needs to act. If it does then it will return true which triggers a log event and then most likely set a value based on it's internal logic. An Ava command to turn on the lights triggers the Ava rules that then find the user and turn on the lights in the room in which the command was uttered a setValue.

Exception Handling

Exception handling is focused on a per rule basis, for the AvaRules it would most likely be an invalidAvaCommand that covered malformed commands or those referencing incorrect data. FridgeRules, CameraRules also have exceptions for malformed statements specialized to their type of data.

Testing

Included with the TestDriver class is a FunctionalityTestDriver that proves the functions of each rule set class and it's exceptions with commands that are meant to be caught or fail.

Risks

This system currently has no authentication leaving a major attack service. Users such as children could close doors and set the status of fire alarms. Also there is no value checking, you could set the state of an object like a TV to Pizza, but it would have no semantic meaning. Additionally the service is a monolith that is unscalable, a service oriented approach may facilitate maintainable scale and moving to a more performant language like Rust or Golang would be prudent.