CSCIE97 Software Design

House Mate Controller Service Design Document

Date: 2017 October 24

Author: Jeremy Clark

Reviewer(s):

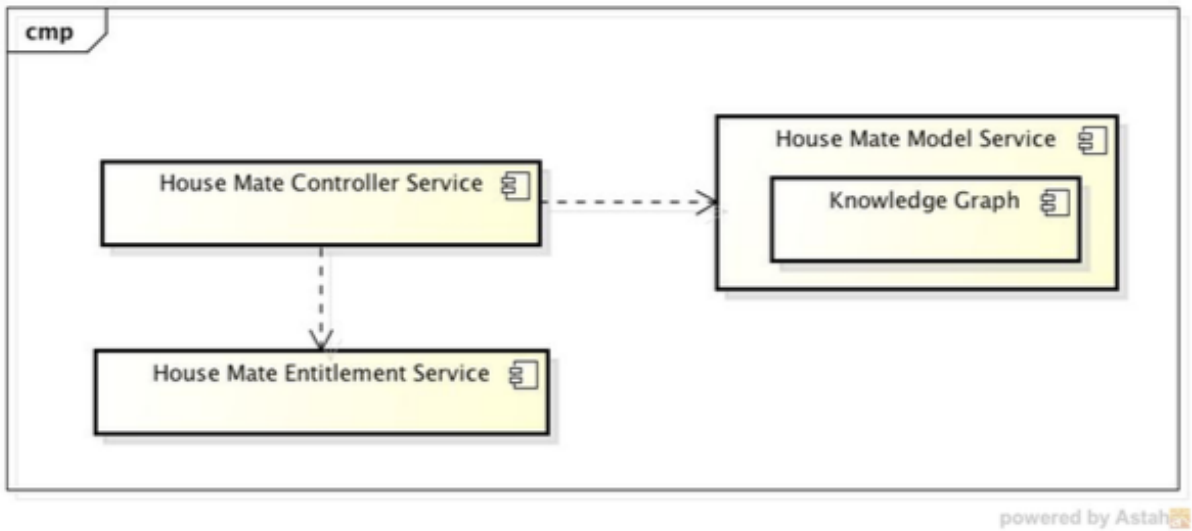
Introduction

This document defines the House Mate Controller 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 Controller Service, is the system component that will listen for input from sensors in the house, update the state of the system, and send any commands to smart devices as a result of the inputs.

The House Mate Controller Service will communicate with the House Mate Model service to query and update state in the House Mate System. As detailed in the House Mate Model Service design document, the modeled classes in the system are:

- House
- Room
- Occupant
- Devices
  - “Appliances”, that have settings/commands
  - “Sensors” that measure/detect the environment
- Measures
- Settings

The controller service will be loaded with “Rules” and “Actions”. Rules will match incoming sensor data (Measures) to see what types of “Actions” might need to be taken.

When a rule is matched, the Controller service may then update the state of “Settings” and “Measures” in the House Mate Model Service to capture the new state of the system.

Settings are properties of smart appliances that, when changed, constitute a ‘command’. For example, a light device may have a “dimness” setting, which, when changed will issue a command to the light to change it’s dimness.

Measures are inputs from smart sensors that will be matched against “Rules” registered in the controller that will update the state of the system in pre-defined ways.

Consider the following: a modeled Ava device might have a Measure called “voice_command” which accepts arbitrary string values. When somebody speaks to Ava, the phrase they utter will be the current value of the “voice_command” Measure. If the controller detects a ‘voice_command’ from Ava which matches a rule, that rule will update the system state via the House Mate Model Service. If the state changes represent a change to the Setting of one of the smart devices, that will manifest as a ‘command’ to be sent to the smart appliance whose Setting was changed.

For example:
Somebody in the kitchen says: “Turn on the ceiling fan”

This might be passed to the controller as:
set sensor House:Kitchen:Ava state voice_command value “Turn on the ceiling fan”
If the controller understands that phrase, a rule will be matched and it will query and update the state of the model:

First it might issue this command to the House Mate Model Service:

```
show House:Room:Kitchen
```

to see if there are any ceiling fan devices registered in the kitchen.

If it finds one, It might issue:

```
set appliance House:Room:Kitchen:CeilingFan status power_mode
value ON
```

And

```
set appliance House:Room:Kitchen:Ava status voice_command value
Turn_On_The_Ceiling_Fan
```

The first command will update the ceiling fan power_mode Setting, and the second command will store a new value for the voice_command Measure, in case the next detected voice_command is something like “Cancel the last command”.

If either of the calls to House Mate Model Service fails, appropriate exception handling will take place - but if they succeed, the controller may then invoke a registered “Action” for the device acting as the receiver of the command. In this case, the Ceiling Fan in the kitchen.

Let’s say there is no ceiling fan in the kitchen; the controller may then issue this command to the HMMS:

```
set appliance House:Room:Kitchen:Ava status voice_response value
“There is no ceiling fan in the kitchen. Did you mean another
room?”
```

This will update the “voice_response” Setting of the Ava appliance, and likely invoke an action to “Speak voice_response” on the device so the Occupant can hear the feedback.

The document that follows will detail the requirements, use cases, and Implementation design that will be used to create the Controller Service. It will include diagrams where helpful, including Activity and Class diagrams to support the Implementation section. There will also be sections describing the testing framework as the Risks associated with this draft of the design.

Requirements

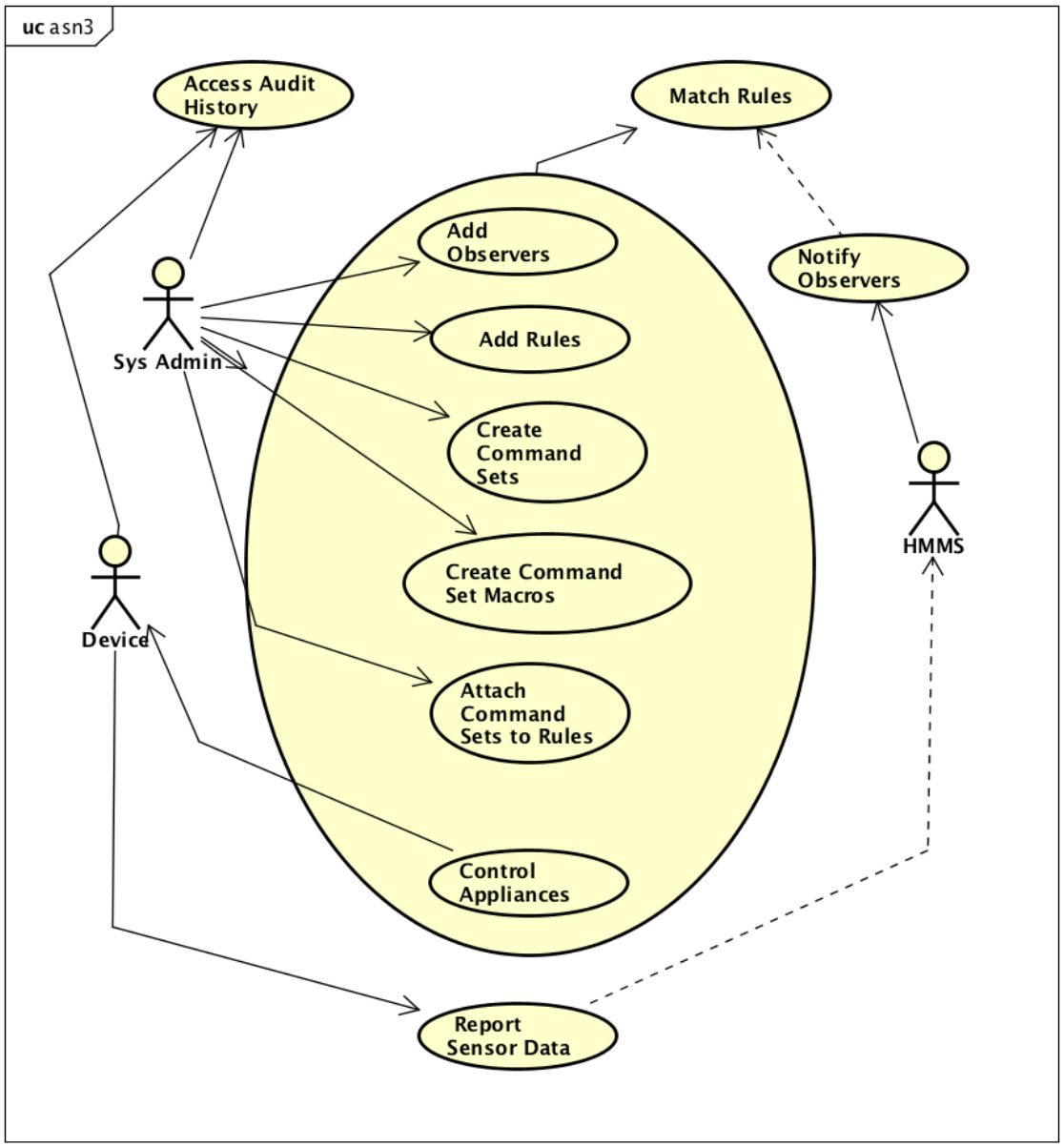
Generally, the House Mate Controller Service is to listen for inputs from smart Devices that are part of the system and apply actions which will change the state accordingly. These state changes may invoke ‘Actions’ or ‘Commands’ that change the physical state of system appliances.

More specifically, it will:

- Allow for registering devices in the system.
- Allow for the dynamic addition of features to a device.
- Observe incoming sensor information, including, but not limited to:
    - Voice commands from intelligent ‘speakers’, like Ava, Google Home, Apple HomePod and the new Sonos speaker.
    - Video signals from cameras, and more specifically motion and facial recognition signals
    - Environmental sensor information like decibels, temperature, brightness, humidity, pressure, smoke and pollutants, water/moisture and more
- Support the creation of “Actions” that will change the state of the system, and invoke physical changes in appliances.
    - A single action can affect more than 1 appliance at a time
- Support the creation of “Rules” that will interpret incoming sensor data, and invoke 1 or more actions.

    - Rules can be matched based on incoming sensor data or by system-invoked state changes
    - Rules can invoke more than one action
    - The actions a rule invokes can trigger a match on more rules that trigger more actions.
- It will log activity in the system, including:
    - All incoming sensor data
    - All matched rules
    - All invoked actions
    - All state changes resulting from actions
- Can be extended to potentially allow for reversal of actions (undo)
- Will communicate exclusively with the House Mate Model Service to read and write system state.

Use Cases


uc asn3
Access Audit History
Match Rules
Add Observers
Notify Observers
Sys Admin
Add Rules
Create Command Sets
HMMS
Create Command Set Macros
Device
Attach Command Sets to Rules
Control Appliances
Report Sensor Data

Use Case Detail

Detect Sensor Information

- Preconditions: The House Mate Model Service (HMMS) has a configuration loaded which includes Devices (Sensors and/or Appliances with sensors)
- Smart sensors throughout the home send data to the House Mate system. The devices are configured to send data directly to the HMMS. The values provided by the sensors represent the ‘facts’ of the system, and they are not subject to interpretation. Here’s the basic flow:
  - The HMMS is configured to add a certain device.
  - That device is configured with the features (Measures and Settings) it supports
  - As the device collects data, it sends the data directly to the HMMS using the HM command interpreter or the HMMS API

Observe Devices

- Precondition: The House Mate Model Service (HMMS) has a configuration loaded which includes Devices (Sensors and/or Appliances with sensors)
- A system admin or device manufacturer determines a ‘Feature’ of interest in one of the available sensor devices (i.e., “I’m interested in whenever the relative humidity changes), and registers an observer function for that feature. A “Feature” in this case is specific measure on a specific device. (For example, the relative humidity measurement of the barometer in the basement). When the Feature changes state, the defined observer function is executed. A device manufacture may provide a ‘bootstrap script’ or a ‘jar file’ or some other sort of archived set of observer functions, while a sys admin may want to add observers on an ad-hoc basis.
- If the Feature doesn’t exist (i.e. there is no barometer in the basement), an error will be returned from the registration request.
- A possible extension would be to register a single observer function for all Features that provide the same measurement: for example, registering a single observer function for all “Temperature” changes in the house - regardless of what device they came from.

Define rules

- Preconditions: Device Observers have been registered in the system
- A system admin or device manufacture defines a ‘rule’ associated with an observer function. An observer could possible have multiple rules associated with it. A rule will

likely apply some conditional checks about the current ‘facts’ present in the observation, like “is this observation within some range” or “does this observation include a voice_command with a known pattern”? It’s also possible that the rule will match every time a device observation is made, regardless of its value or measurement.

Define Reusable Commands Sets

- Preconditions: A command language or plug-in architecture is available.
- A device manufacturer or System Admin can create command sets (consisting of one or more commands) that can be attached to rules. As noted, a command set can be as little as 1 command. These command sets are meant to be ‘reusable’ across rules - for example, a command set might turn off the lights and lower the temperature in a room. We may want to execute that command set either when everybody leaves the room or when its determined that somebody has fallen asleep on the couch.

Define macro command sets

- Prerequisites: Command sets exists
- Effectively, a command set of command sets. A system admin or device manufacturer can organize existing command sets into bigger command sets with more functionality.
- Possible extension will allow an arbitrary number of nested command sets.

Attach/Detach command sets to rules

- Preconditions: Rules have been defined
- A device manufacturer or System Admin can associate command sets with matched rules. If the command set no longer makes sense for a particular rule, it can be removed from the rule.

Match Rules and Execute Commands

- When a rule is matched, the controller will execute the defined command sets within the rule. Keep in mind that a command will likely change state, which could trigger more rules which could execute more command sets, which could lead to circular execution. The controller must be responsible for detecting and handling circular command execution loops.
- Specifically, we may want to listen for a certain voice command and then update an associated device. For example a voice command of “Dim Lights” should dim the lights in the room where the voice command was issued.

Send Commands to appliances

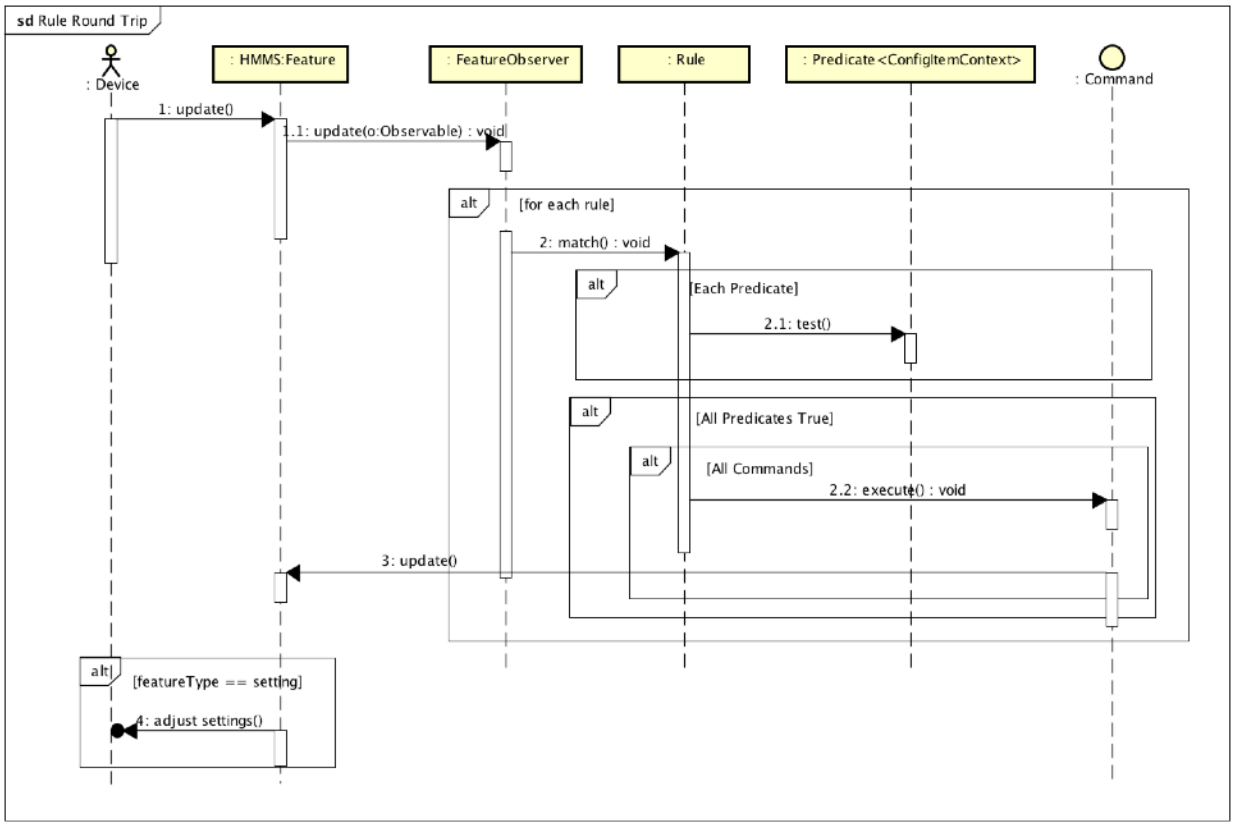
- As part of the command executions, the controller may update appliance state - which will materialize as a ‘command’ sent to an appliance. This will be the ‘tangible-manifestation’ of the automated home from the occupant’s perspective.

Audit System History

- Prerequisites: The system is activated and activity has taken place
- A system admin or device has access to the history of activity in the system. Initially this will include
  - A history of all configuration changes
  - A log of registered rules and commands
  - A log of all device state changes
  - A history of all occupant history
- This will be used for troubleshooting, status reporting, investigation, and possibly for the support of ‘undoable’ actions.
- Possible extension: It can also be used to ‘reset’ the state of the system in case of power outage or other system failure.

Activity Diagrams

This diagram illustrates an “Update -> Rule -> Command” round trip. The classes represented here are described in more detail in the Implementation section.



Implementation

The key constructs in the controller are:

- Feature Observers
- Rules
- Predicates
- Contexts
- Commands

Generally speaking:

Feature Observers listen to House Mate Model Features (see House Mate Model Design document for more details) for changes.

Rules are attached to Feature observers, and when an observer is notified all attached rules are “Matched”.

Contexts are attached to Rules, and provide the data elements that will be tested by predicates.

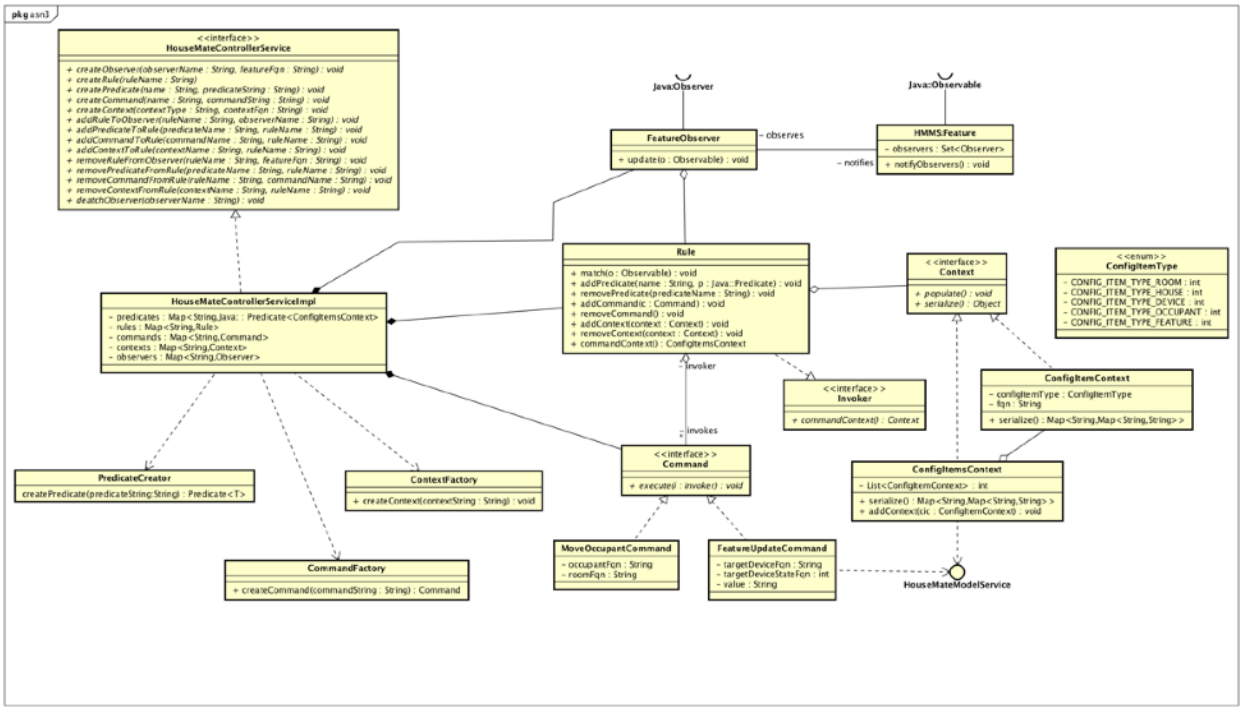
Predicates are attached to Rules. Predicates are simple tests against a set of data.

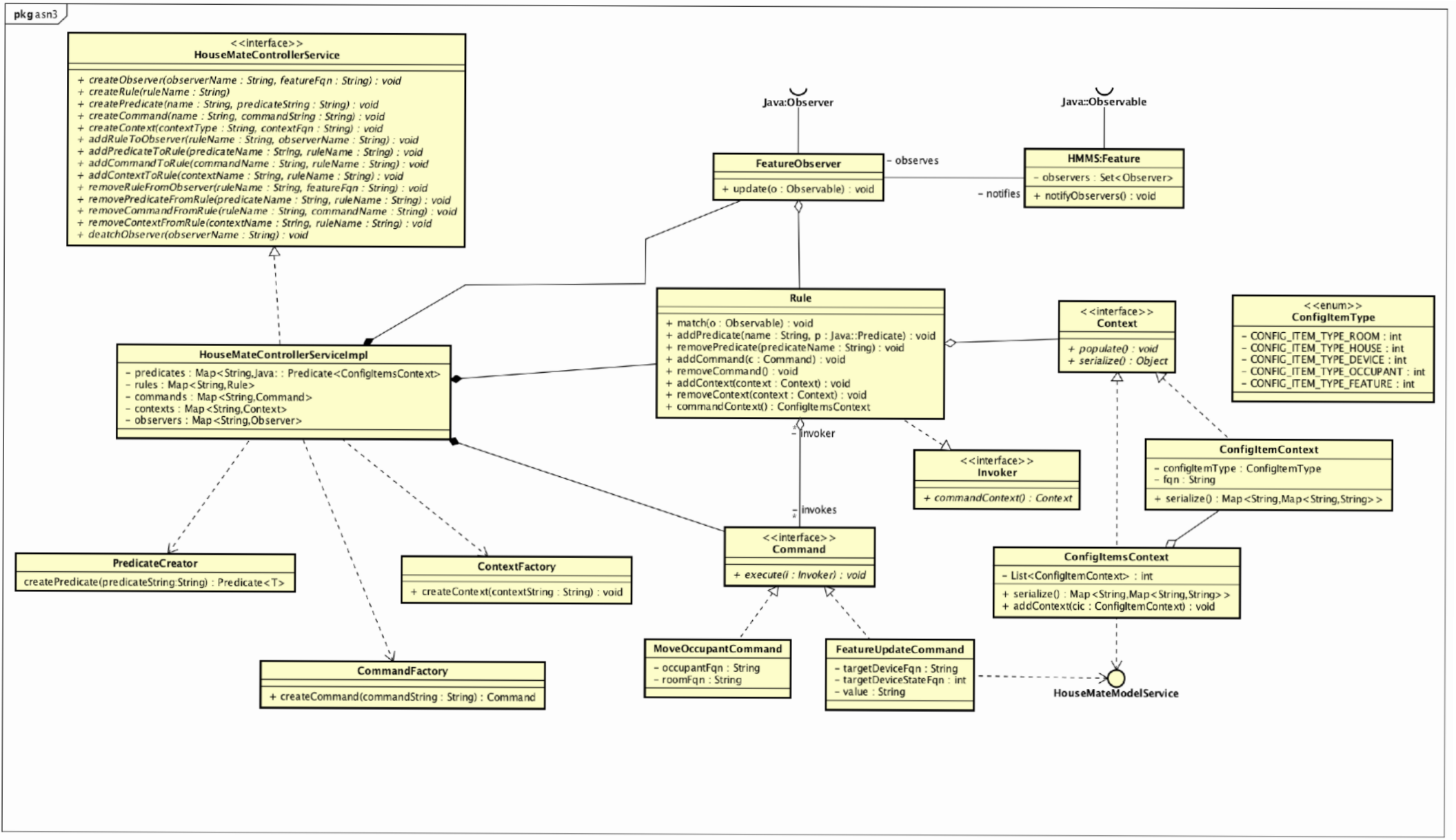
When a rule’s match() method is called, all predicates are provided with the rule’s contexts, and tested. If all predicates test positive, then…

Commands that are attached to the rule are executed. Commands are passed an Invoker object which will maintain its own context; in concrete terms, the Rule object is the Invoker and it has “Context” that can be used by the Command). The commands will maintain the target of the command (for example a device feature) as well as a value to pass to the target. Sometimes that value will be derived from the context available in the Invoker.

The diagrams and class dictionary that follow provide detail on these concepts.

Class Diagram




pkg asn3
<<interface>>
HouseMateControllerService
+ createObserver(observerName : String, featureFqn : String) : void
+ createRule(ruleName : String)
+ createPredicate(name : String, predicateString : String) : void
+ createCommand(name : String, commandString : String) : void
+ createContext(contextType : String, contextFqn : String) : void
+ addRuleToObserver(ruleName : String, observerName : String) : void
+ addPredicateToRule(predicateName : String, ruleName : String) : void
+ addCommandToRule(commandName : String, ruleName : String) : void
+ addContextToRule(contextName : String, ruleName : String) : void
+ removeRuleFromObserver(ruleName : String, featureFqn : String) : void
+ removePredicateFromRule(predicateName : String, ruleName : String) : void
+ removeCommandFromRule(ruleName : String, commandName : String) : void
+ removeContextFromRule(contextName : String, ruleName : String) : void
+ deatchObserver(observerName : String) : void
Java::Observer
Java::Observable
FeatureObserver
+ update(o : Observable) : void
- observes
- notifies
HMMS:Feature
- observers : Set<Observer>
+ notifyObservers() : void
Rule
+ match(o : Observable) : void
+ addPredicate(name : String, p : Java::Predicate) : void
+ removePredicate(predicateName : String) : void
+ addCommand(c : Command) : void
+ removeCommand() : void
+ addContext(context : Context) : void
+ removeContext(context : Context) : void
+ commandContext() : ConfigItemsContext
<<interface>>
Context
+ populate() : void
+ serialize() : Object
<<enum>>
ConfigItemType
- CONFIG_ITEM_TYPE_ROOM : int
- CONFIG_ITEM_TYPE_HOUSE : int
- CONFIG_ITEM_TYPE_DEVICE : int
- CONFIG_ITEM_TYPE_OCCUPANT : int
- CONFIG_ITEM_TYPE_FEATURE : int
HouseMateControllerServiceImpl
- predicates : Map<String,Java: : Predicate<ConfigItemsContext>
- rules : Map<String,Rule>
- commands : Map<String,Command>
- contexts : Map<String,Context>
- observers : Map<String,Observer>
- invoker
<<interface>>
Invoker
+ commandContext() : Context
ConfigItemContext
- configItemType : ConfigItemType
- fqn : String
+ serialize() : Map<String,Map<String,String>>
- invokes
<<interface>>
Command
+ execute(i : Invoker) : void
PredicateCreator
createPredicate(predicateString:String) : Predicate<T>
ContextFactory
+ createContext(contextString : String) : void
ConfigItemsContext
- List<ConfigItemContext> : int
+ serialize() : Map<String,Map<String,String>>
+ addContext(cic : ConfigItemContext) : void
CommandFactory
+ createCommand(commandString : String) : Command
MoveOccupantCommand
- occupantFqn : String
- roomFqn : String
FeatureUpdateCommand
- targetDeviceFqn : String
- targetDeviceStateFqn : int
- value : String
HouseMateModelService

Class Dictionary

Interface::HouseMateControllerService

Interface

Public API that allows access to the House Mate Controller 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 **createObserver**(observerName : String, featureFqn : String) : void	Adds an observer to a Feature in the House Mate Model.	ItemNotFoundException UnauthorizedException
public **createRule**(ruleName : String)	Creates a Rule that can be attached to an observer. Predicates, Context and Commands will be attached to a rule. Adds created rule to the managed collection of Rules.	ItemExistsException UnauthorizedException
public **createPredicate**(name : String, predicateString : String) : void	Creates a Predicate and adds it to the collection of predicates. A predicate is a snippet of code that will test a condition in for a given context. The predicates will be attached to rules, and tested based on the context associated wit that rule. The predicate string can be any snippet of code that is supported by the Predicate Factory. Initially, the only supported predicate string type will be snippets of JavaScript that will accept ConfigItemContext objects as their parameters.	InvalidPredicateString ItemExistsException UnauthorizedException

Operations		
Operation	**Description**	**Throws**
public **createCommand**(name : String, commandString : String) : void	Adds a Command to the controller. The commandString is a valid command string that will be supported by the Command Factory to create a command. The commands are attached to Rules and executed when all Predicates attached to the rule are true. Initially, the only supported command string type will be HouseMateModelService::CommandInterpreter command strings.	InvalidCommandString ItemExistsException ImportException UnauthorizedException
public **createContext**(contextType : String, contextString : String) : void	Creates a context and adds to the context collection. Contexts are added to Rules to provide facts to the Predicates. The Context object created will have a getContext() method that will return facts that can be used by predicates. The contextString can be any string that the ContextFactory knows how to produce. The contextString could be a “sql” query or a webservice URL. In the HMCS, the context objects will use HMMS FQNs to find facts in the KnowledgeGraph that will be assessed by the predicates to determine whether system state dictates further action. Initially, the only valid contextString will be a House Mate Model FQN.	InvalidContextString ItemNotFoundException ImportException UnauthorizedException
public **addRuleToObserver**(ruleName : String, observerName : String) : void	Attaches a rule to an observer, so that the the Rule’s match() method is invoked when the observer is notified.	ItemNotFoundException UnauthorizedException
public **addPredicateToRule**(predicateName : String, ruleName : String) : void	Attaches a Predicate to a Rule so that the predicate.test() will be called during the Rules match routine. All attached predicates will be tested.	ItemNotFoundException UnauthorizedException
public **addCommandToRule**(commandName : String, ruleName:String) : void	Adds a Command to a Rule. If all predicates are true in a rule, ALL attached commands will be executed.	ItemNotFoundException UnauthorizedException

Operations		
Operation	**Description**	**Throws**
public **addContextToRule**(contextName : String, ruleName : String) : void	Adds a Context object to a Rule. The result of the Context.getContext() method will be passed to Predicates as facts to test the Predicates against.	ItemNotFoundException UnauthorizedException
public **removeRuleFromObserver**(ruleName : String, featureFqn : String) : void	Removes a rule from an observer.	ItemNotFoundException UnauthorizedException
public **removePredicateFromRule(**predicateName : String, ruleName : String) : void	Removes a Predicate from a rule.	ItemNotFoundException UnauthorizedException
public **removeCommandFromRule**(ruleName : String, commandName : String) : void	Removes a command from a rule.	ItemNotFoundException UnauthorizedException
public **removeContextFromRule**(contextName : String, ruleName : String) : void	Removes a context from a rule	ItemNotFoundException UnauthorizedException
public **deatchObserver**(observerName : String) : void	Detaches an observer from the Feature its attached to.	ItemNotFoundException UnauthorizedException

---

Class::HouseMateControllerServiceImpl

Implements Interface::HouseMateControllerService

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:

Associations		
Association	**Type**	**Description**
observers	Map<String, Observer>	The observers registered in the Controller
rules	Map<String, Rule>	The list of occupants associated with the service
commands	Map<String,Command>	The list of commands available in the controller.
contexts	Map<String,Context<T>>	The list of context available in the controller.
predicates	Map<String,Predicate>	The list of predicates available in the controller.

Class::FeatureObserver

Implements Java::Observer

Observes a Feature object, as defined in the HouseMateModel. The Feature object will be enhanced to implement the “Observable” java interface.

Operations		
Operation	**Description**	**Throws**
public **update**() : void	Will be called by objects being observed, which in this case are “Feature” objects. When the Feature is updated and “notifyObservers” is called, this will be invoked and execute the match() function on all associated rules.	

Associations		
Association	**Type**	**Description**
rules	Map<String, Rule>	The rules to be matched when the observer is notified.

Interface::Invoker

Represents an ‘Invoker’ of a command. The “Command::execute()” method will take an Invoker as a parameter. Invokers implement 1 method “commandContext()”, which will return an object that implements Context, so the command can have some data if the invoker chooses to provide some.

Operations		
Operation	**Description**	**Throws**
public **commandContext**() : Context	Returns a “Context” object (see Interface::Context) which can provide data to the invoked command.	

Class::Rule

Implements Interface::Invoker

Represents a ‘rule’ that tests system state, and given certain conditions will execute commands to update devices. Will be composed of Predicates, Contexts and Commands.

Operations		
Operation	**Description**	**Throws**
public **match**() : void	Performs the match routine, which will populate contexts, test predicates and execute commands if all predicates return true.	
public **addPredicate**(name : String, p : Predicate<T>) : void	Adds a predicate to the rule.	ItemExistsException

Operations		
Operation	**Description**	**Throws**
public **removePredicate**(predicateName : String) : void	Removes a predicate from the rule	ItemNotFoundException
public **addCommand**(c : Command) : void	Adds a command to the rule.	ItemExistsException
public **removeCommand**(commandName : String) : void	Removes a command from the rule	ItemNotFoundException
public **addContext**(context : Context<T>) : void	Adds a context to the rule	ItemNotFoundException
public **removeContext**(context : Context<T>) : void	Removes a context from the rule	ItemNotFoundException UnauthorizedException
public **commandContext**() : ConfigItemsContext	Override of Invoker interface method. Will return a ConfigItemsContext object when called.	

Associations		
Association	**Type**	**Description**
contexts	Map<String, Context>	The list of contexts associated with the rule
predicates	Map<String, Predicate>	The list of predicates associated with the rule
commands	Map<String,Command>	The list of commands associated with the rule

---

Interface::Command

A command interface, exposing a single method: execute().

Operations		
Operation	**Description**	**Throws**
public **execute**(i : Invoker) : void	Called to execute the command. Actual command behavior will be defined by implemented by the implementors. Takes an Invoker object as a parameter so the invoker's commandContext() method can be called to possible inform the execution behavior.	ItemNotFoundException

---

Class::FeatureUpdateCommand

Implements Interface::Command

This class encapsulates the data needed to perform a Feature update in the HouseMateModelService. One of these objects will be created for a command with a pre-known discrete value. For example, to create a "Turn Oven Off" command. This would be constructed with a final value of "OFF".

Attributes		
Attribute	**Description**	**Throws**
- targetDeviceFqn : String	The Device hosting the feature to be updated. This would be something like "House:LivingRoom:Thermostat"	
- targetDeviceStateFqn : String	The DeviceState fan of the Feature being updated. This would be something like "Setting:Temperature"	
- value : String	The target value for feature update, OR a javascript string that can be used to	

Operations		
Operation	**Description**	**Throws**
public **execute**(i : Invoker) : void	If the value is a compilable javascript string, the execute() method will call fetch the passed in Invoker's context and run the javascript to derive the update value. Otherwise, will use the value attribute directly and call updateFeature(deviceFqn, deviceStateFqn, value) in the HouseMateModelService	ItemNotFoundException

---

Class::MoveOccupantCommand

Implements Interface::Command

This class encapsulates the data needed to perform a moveOccupant() command in the HMMS.

Attributes		
Attribute	**Description**	**Throws**
- occupantFqn : String	The occupant to be moved.	
- roomFqn : String	The destination room	

Operations		
Operation	**Description**	**Throws**
public **execute**() : void	Is basically a wrapper around "moveOccupant(token, occupantFqn, roomFqn)"	ItemNotFoundException

Interface::Context

Interface that provides methods for accessing data ‘Contexts’ that can be used with Predicate objects. Has a generic type parameter that will be provided by implementors that determines what kind of data will be returned from “getContext()”

Operations		
Operation	**Description**	**Throws**
public **populate**() : void	Will populate the context object with data based on its configuration.	
public **serialize**() : Object	Returns the populated context data. The Object return types will be made concrete by the subclasses.	

ConfigItemContext

Implements Interface::Context

Provides context fetched from the HouseMateModel, by being configured with a ConfigItem concrete type and an FQN. These context objects will be associated with rules, populated and passed to predicates during the rule’s match() operation.

Attributes		
Attribute	**Description**	**Throws**
- configItemType : String	The concrete type of ConfigItem from the House Mate Model.	
- fqn : String	The FQN of the item for which to fetch context.	

Operations		
Operation	**Description**	**Throws**
public **populate**() : void	Based on the configItemType will call the appropriate “show” method in the HouseMateModelService	ItemNotFoundException
public **serialize**() : <Map<String,Map<String,String>>>	Will return all Triples fetched from the HouseMateModelSerivce as a 2-D map, where the top level keys are all subjects in the context and the values are a list of predicate/object pairs for that subject. For example: device1:Temperature is_measuring 75 device1:Setting1 is_set_to something, would be: `{'device1:Temperature': {'is_measuring':75}, device1:Setting1': {'is_set_to':something}},`	

ConfigItemsContext

Implements Interface::Context

Basically a collection of ConfigItemContex objects. But it implements the Context interface so that the “Serialize” method will return serialized data for every associated ConfigItemContext obejct.

Attributes		
Attribute	**Description**	**Throws**
- contexts : List<ConfigItemContext>	A list of ConfigItemContext objects	

Operations		
Operation	**Description**	**Throws**
public **serialize**() : <Map<String,Map<String,String>>>	Will call serialize() on all of items in ‘contexts’ and merge them together into a <Map<String,Map<String,String>>>	ItemNotFoundException

Operations		
Operation	**Description**	**Throws**
public **serialize**() : <Map<String,Map<String,String>>>	Will return all Triples fetched from the HouseMateModelSerivce as a 2-D map, where the top level keys are all subjects in the context and the values are a list of predicate/object pairs for that subject. For example: device1:Temperature is_measuring 75 device1:Setting1 is_set_to something, would be: `{'device1:Temperature': {'is_measuring':75}, device1:Setting1': {'is_set_to':something}},`	

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.

InvalidCommandString

Thrown when trying to create a command with an invalid command string.

Attributes	
Attribute	**Description**
private **commandString** : String	The fully qualified name of the device

---

InvalidPredicateString

Thrown when trying to create a predicate with an invalid predicate string.

Attributes	
Attribute	**Description**
private **predicateString** : String	The passed in predicate string

InvalidContextString

Thrown when trying to create a Context with an invalid context string.

Attributes	
Attribute	**Description**
private **contextString** : String	The passed in context string

Implementation Details

The implementation will support a dynamically configurable controller system. For the puposes of this phase the controller configuration will only be stored in memory and not persisted to disk. The configurations will not be stored to the knowledge base.

Modifications to the HouseMateModelService

Added new public API call:

```
+List<String> getFeature(String authToken, String featureFqn).
```

This behaves just like the getFeature(String authToken, String deviceFqn, String deviceStateFqn); except it takes a fully qualified Feature FQN as a single argument.

A note on Predicates

One key piece of the implementation for this design is how the Predicates will be created. For now, they will be created by a Predicate creator that will return Java platform Predicates of type Predicate<Map<String, Map<String,String>>>.The predicate

creator will take the JavaScript passed in the Controller Service ‘createPredicate()’ execute that JavaScript in the JavaScript ScriptEngine. The engine will be passed the Predicate test parameter (the Map<String, Map<String, String>>) object and use standard javascript style . notation to read and test the context data.

A rule will take any Predicate type, as there is a clear case to create different types of predicates that use native Java code. For example, the system could be enhanced to read in plug-in jar files that contain native java predicates and use those instead of or in addition to JavaScript predicates. I’m going with the JavaScript style for now in order to make the system flexible and configurable.

A note on Logging

All key events will be logged for auditing and replay/undo possibilities. This includes:

- Feature Updates
- Observer Updates
- Rule match() executions
- Predicate tests (with context information)
- Command executions

This implementation will simply use System.out to perform console logging.

Testing

Unit Testing

Use JUnit test classes for each concrete class. Unit tests will be executed @ build time. This is made pretty easy by using IntelliJ.

Command/Integration Testing

Implement a test driver class called ControllerTestDriver that implements a static main() method. The main() method accepts a script file that has command lines that include commands to bootstrap the Model with houses/rooms/devices/etc. and commands to setup the controller rules, predicates, etc.

Subsequent to the setup, commands mimicking the updating of sensor and setting data will tigger rules to be executed and commands invoked.

All of the output will be logged and stored for review, so that the trace of operations can be audited.

Risks

- Accepting raw (JS) code as the predicate code makes the system subject to injection attacks.
- Rules could trigger an infinite loop of observe -> match -> update cycles if not careful.
- Referring to controller configuration objects by “Name” is not the most referentially sound method.