Università degli studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Progetto di Ingegneria del Software

SmartHome

Gianlorenzo Occhipinti	Christian Tasca	Davide Messina
829524	830187	830241

Milano, Gennaio 2020

Indice

Introduzione

Il principale obiettivo per la progettazione di questo software è stato quello di creare una logica che gestisse in toto le sfaccettatissime situazioni possibili all'interno di un ambiente domestico, senza introdurre restrizioni a priori, come la creazione di un ambiente "standard" da gestire, ma tenendo il più possibile l'esperienza versatile e personalizzabile: questo comporta ovviamente la necessità di gestire situazioni di concorrenza con tutte le conflittualità che ne derivano, tipiche di un sistema che segue una logica ad eventi (come è appunto un "cervello domotico").

Tra le specifiche più interessanti spicca la capacità del sistema di classificare due tipi di situazioni: quelle che possono essere completamente risolte tramite automazioni (automatiche) e quelle che necessitano invece di un intervento dell'utente (semi-automatiche). Infatti il software, una volta impostato, si occupa autonomamente della maggior parte delle situazioni, a meno che non sovvengano materie di preferenza sconosciuta.

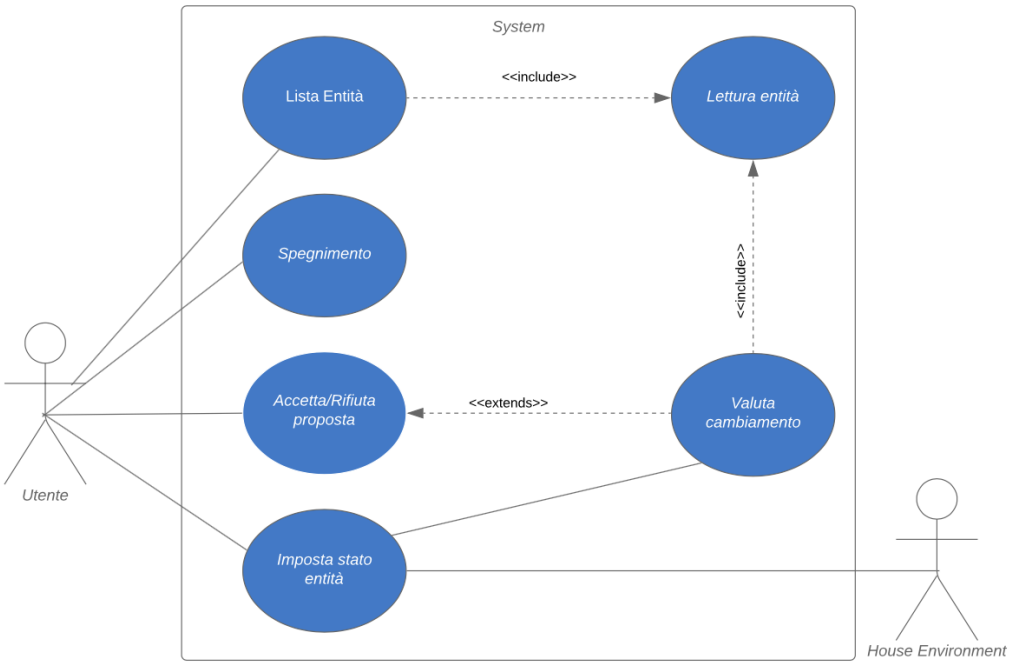
L'utente può impostare come meglio preferisce la gestione dell'ambiente domestico tramite la modifica di un file JSON che ne contiene le automazioni specifiche ad ogni situazione, suddivise in condizioni che si devono verificare e conseguenze che ne derivano.

Un'altra caratteristica è la registrazione dei dati rilevati all'interno dell'ambiente domestico, che "preparano il terreno" per una possibile estensione del software per funzionalità di analisi statistica dei dati, stima delle preferenze dell'utente, capacità di machine learning e altre ancora.

1. Analisi

1.1 Casi d'uso

L'analisi dei casi d'uso riguarda lo studio degli scenari che rappresentano l'interazione di entità esterne al sistema, gli attori, con il sistema stesso o sue interazioni. Questo significa che nei casi d'uso non vengono prese in considerazione le azioni interne al sistema, ma solo il modo in cui gli attori possono interagire con esso.



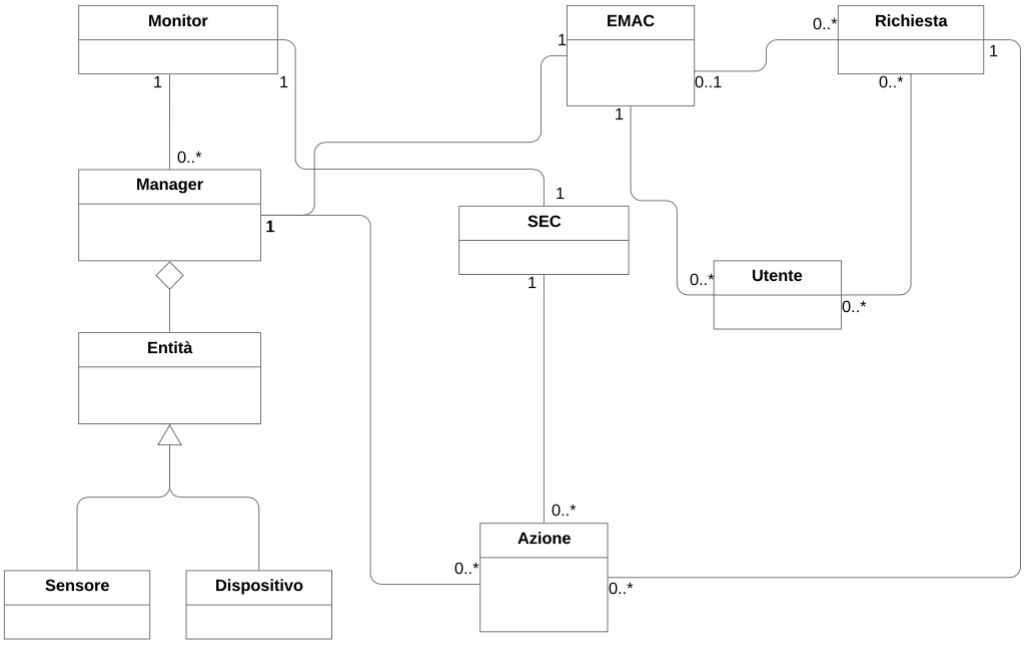
Nel caso d'uso rappresentato l'attore *utente* può eseguire le seguenti operazioni:

- Spegnere il sistema;
- Richiedere la lista delle varie entità presenti all'interno del sistema;
- Accettare delle azioni proposte dal sistema, infatti in base a ciò che l'utente inserisce nelle varie automazioni, il sistema prima di eseguire delle azioni può chiedere all'utente l'approvazione, come per esempio per aprire la finestra nel caso la temperatura in casa sia troppo elevata;
- Richiedere all'utente di eseguire una certa operazione, come per esempio far spegnere una luce.

L'attore *House Environment* può essere rappresentato come un sistema esterno, una sorta di simulatore che tiene traccia di tutti i sensori e dispositivi presenti: infatti esso può cambiare e impostare lo stato interno delle entità. Per esempio quando la temperatura aumenta, *House Environment* dovrà settare lo stato delle entità interne al sistema.

1.2 Diagramma delle classi di dominio

L' obiettivo del diagramma delle classi di dominio è quello di descrivere le informazioni della realtà di interesse secondo una rappresentazione concettuale e ad oggetti.

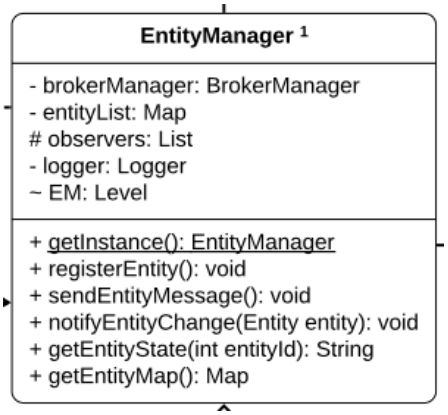


La classe *Entità* contiene tutte le entità che possono essere presenti all'interno del nostro sistema, queste si dividono in sensori e dispositivi. Il *sensore* è un entità che può essere solamente letta il cui stato/valore non può essere modificato manualmente, come per esempio un sensore di temperatura. Al contrario lo stato di un *dispositivo* può essere modificato, ne sono un esempio le finestre automatizzate che in base ai vari eventi presenti nel sistema, possono chiudersi/aprirsi.
La *classe Manager* è una collezione di Entità che si occupa della gestione delle stesse, dalla modifica alla notifica del cambiamento di una entità a EMAC e a Monitor. Il *monitor* si occupa di archiviare i vari stati delle entità e di effettuare delle analisi su di essi.
L' Entity Monitor Automation Control (EMAC) si occupa di monitorare i cambiamenti delle varie entità e di verificare se il cambiamento comporta delle modifiche automatiche da effettuare, un esempio può essere quando il sensore conta persone passa da 1 a 0, in questo caso l'EMAC deve controllare lo stato delle luci e nel caso spegnerle se risultano accese.
Le *richieste* sono un insieme di *azioni* che devono essere eseguite.
Il Secutity and Error Coverage (SEC) si occupa della sicurezza ma cosa più importante dei conflitti che possono generarsi con le varie richieste. Viene a crearsi un conflitto quando il sistema deve aprire le finestre a causa di una perdita di gas, ma successivamente, a causa del crollo della temperatura, l'EMAC crea una richiesta per chiudere le finestre: questa richiesta non può essere però compiuta, in quanto il sensore di gas rileva ancora una quantità di gas superiore alla norma.

2. Progettazione

2.1 Diagramma delle classi di progetto

2.1.1 EntityManager



La classe *EntityManager* possiede diversi attributi tra cui:

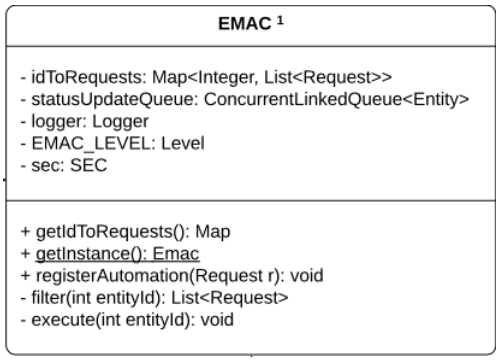
- *brokerManager*: è un oggetto di tipo BrokerManager, che viene utilizzato per effettuare il collegamento con il simulatore.
- *entityList* è un oggetto di tipo Map, ovvero una mappa contenente tutte le informazioni per ogni singola entità.
- *observers* è un oggetto di tipo List, contenente tutte le classi che si sono "iscritte" per essere notificate di un cambiamento di stato.

I metodi principali sono:

- *getInstance()*: permette a chi lo richiama di avere un istanza di EntityManager, infatti essendo una classe singleton(5.2.2), non è possibile istanziarla.
- *sendEntityMessage()* viene richiamato quando vogliamo cambiare lo stato dell'entità passata come parametro.
- *notifyEntityChange(Entity entity)* viene richiamata per notificare alla lista di *observers* un cambiamento avvenuto nell'entità passata come parametro.

Inoltre EntityManager implementa l'interfaccia Subject che fa parte del design pattern observer(5.2.1).

2.1.2 EMAC



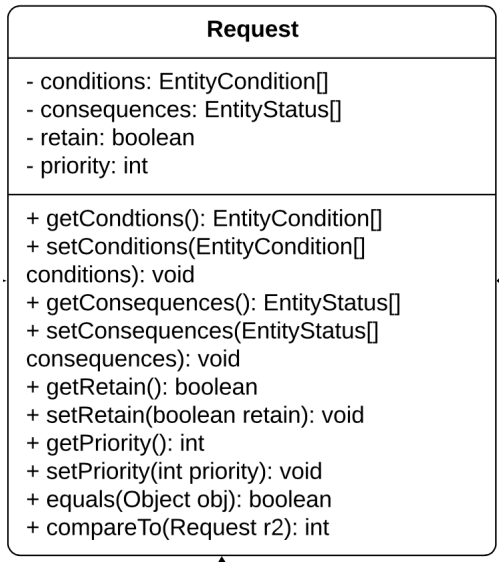
Gli attributi principali di EMAC sono:

- *idToRequests* è un oggetto di tipo Map, che dato un intero restituisce l'entità associata ad esso.
- *statusUpdateQueue* è una lista che contiene le entità che devono essere controllate. Questo attributo è necessario per evitare la concorrenza tra diversi Thread.

I metodi sono:

- *registerAutomation(Request r)* è un metodo che viene utilizzato per registrare le varie automazioni inserite nel file Automations.json dall'utente.
- *filter()* si occupa di filtrare le richieste valide tra le richieste associate all'entità che è identificata dall'id passato come parametro.
- *execute()* esegue le richieste valide (verificate tramite il metodo filter) in base alla priorità che hanno.

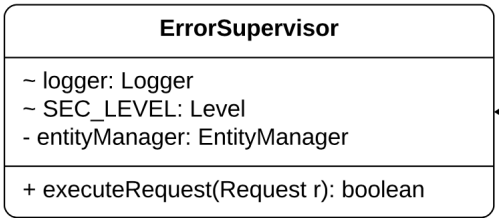
2.1.3 Request



Request ha come attributi:

- *conditions[]* è un array di tipo EntityCondition, che ha al suo interno un insieme di condizioni che devono essere verificate prima di eseguire la richiesta.
- *consequences[]* è un array di tipo EntityStatus che contiene tutte le azioni che devono essere compiute.
- *retain* è un attributo boolean, che è settato su true se la sua esecuzione deve rimanere "attiva" finché le condizioni di conditions[] sono verificate.
- *priority* è un numero intero che indica il livello di priorità della singola richiesta.
- *askPermission* è un attributo boolean, che è settato su true se prima di effettuare la richiesta deve essere richiesto il permesso all'utente.

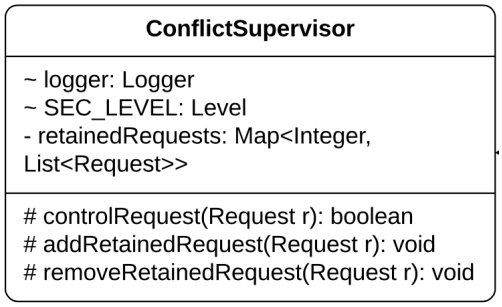
2.1.4 ErrorSupervisor



I metodi principali sono:

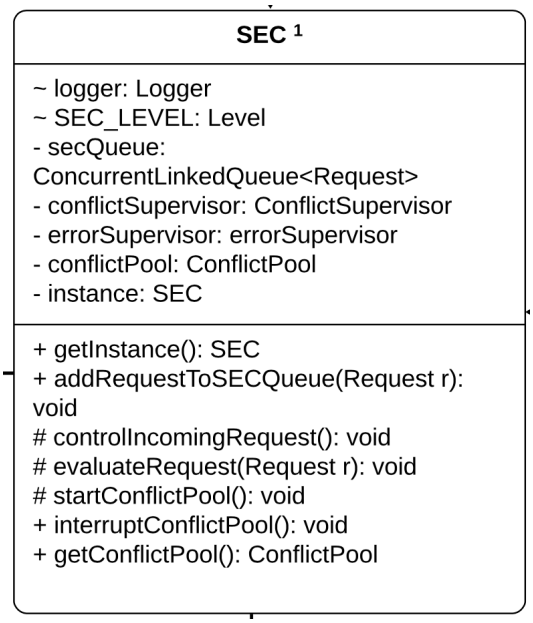
- *executeRequest()* si occupa di verificare se una richiesta prima di essere eseguita deve essere accettata dall'utente.
- *sendMessagesToEntities()* si occupa della gestione delle eccezioni generate dal cambio di stato delle varie entità, per esempio se si cercasse di impostare uno stato non valido.

2.1.5 ConflictSupervisor



Come attributo troviamo *retainedRequests* che è un oggetto di tipo mappa, che, preso un numero intero (ID entità), associa ad esso una lista di richieste. Queste sono le richieste retained, ovvero che devono rimanere attive finché non si verifica una certo cambiamento: per esempio devo mantenere le finestre aperte fino a quando il sensore del gas non rientra nei limiti stabiliti. Mentre il metodo controlRequest() si occupa di verificare eventuali conflitti con azioni ancora presenti, restituisce true nel caso si possa procedere alla richiesta e restituisce false nel caso in cui invece si entri in conflitto.

2.1.6 SEC

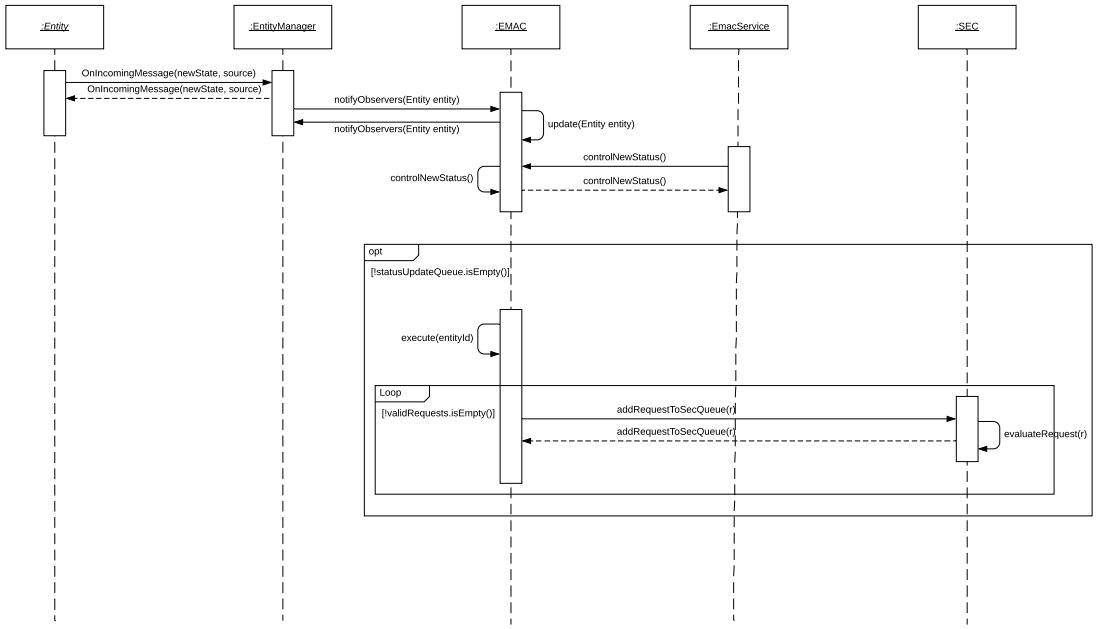


SEC si occupa di gestire i conflitti e di eseguire le richieste; i metodi principali sono:

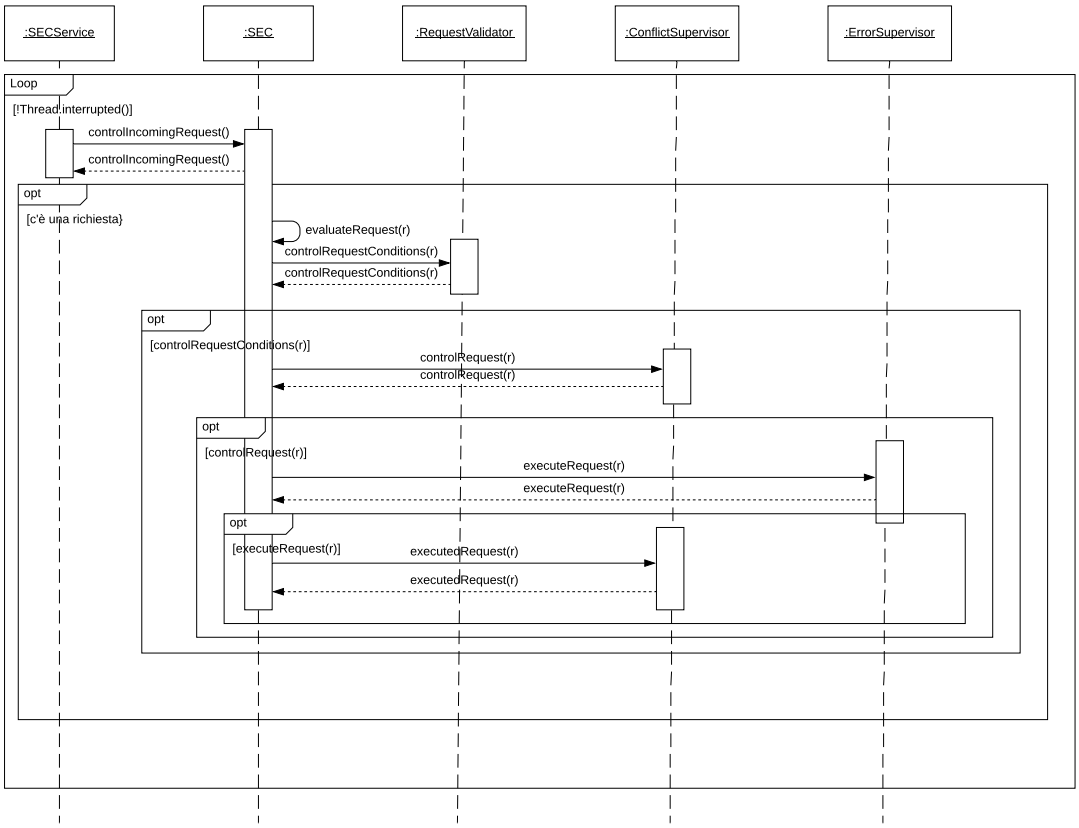
- *evaluateRequest()* si occupa di verificare che una richiesta non abbia conflitti con altre tramite il metodo controlRequest()(2.1.5), nel caso non ci siano conflitti richiama il metodo executeRequest()(2.1.4) di ErrorSupervisor; al contrario se risultano conflitti, la richiesta viene aggiunta alla conflictPool.

2.2 Diagramma di sequenza

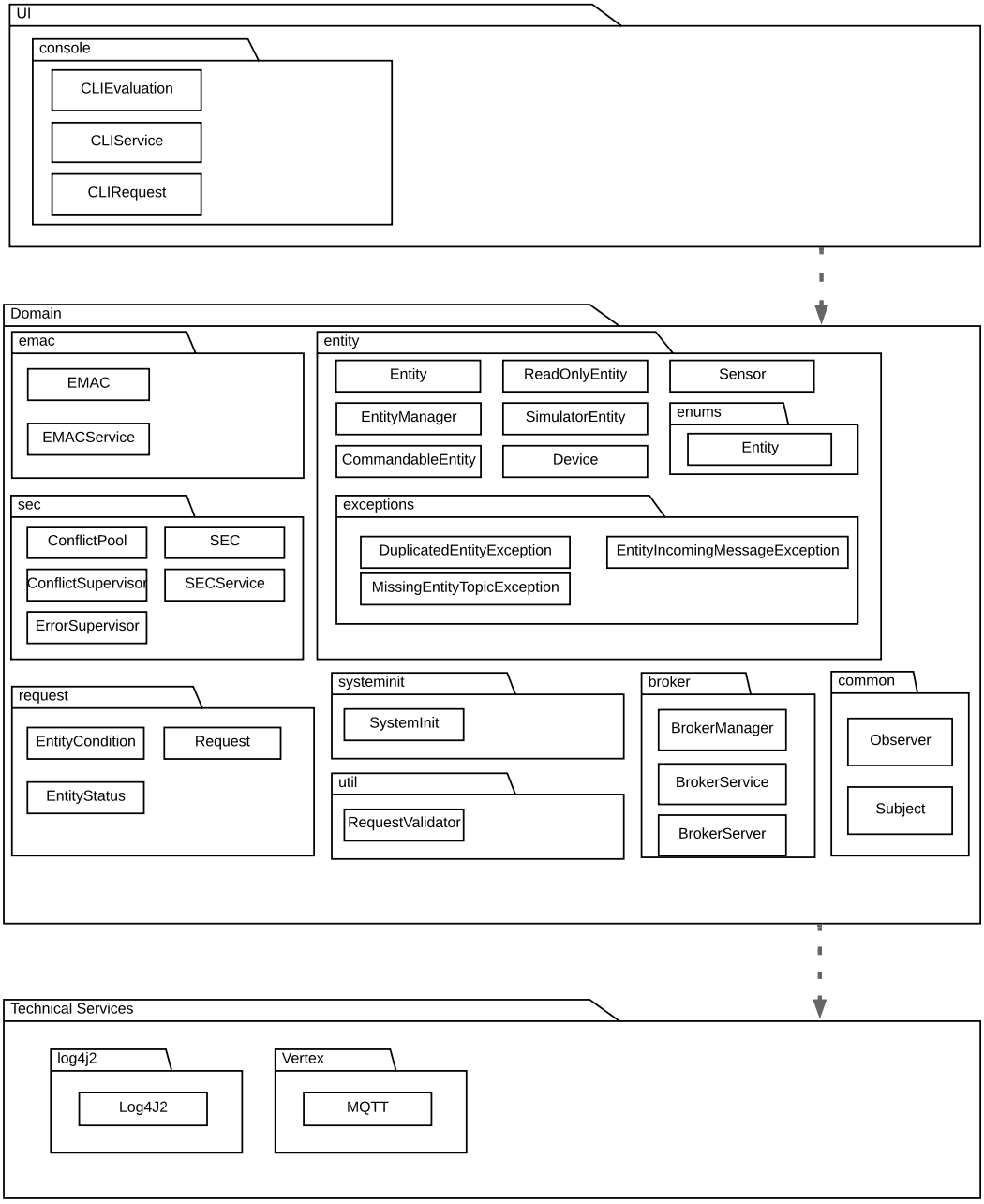
2.2.1 Cambio di stato di un'entità



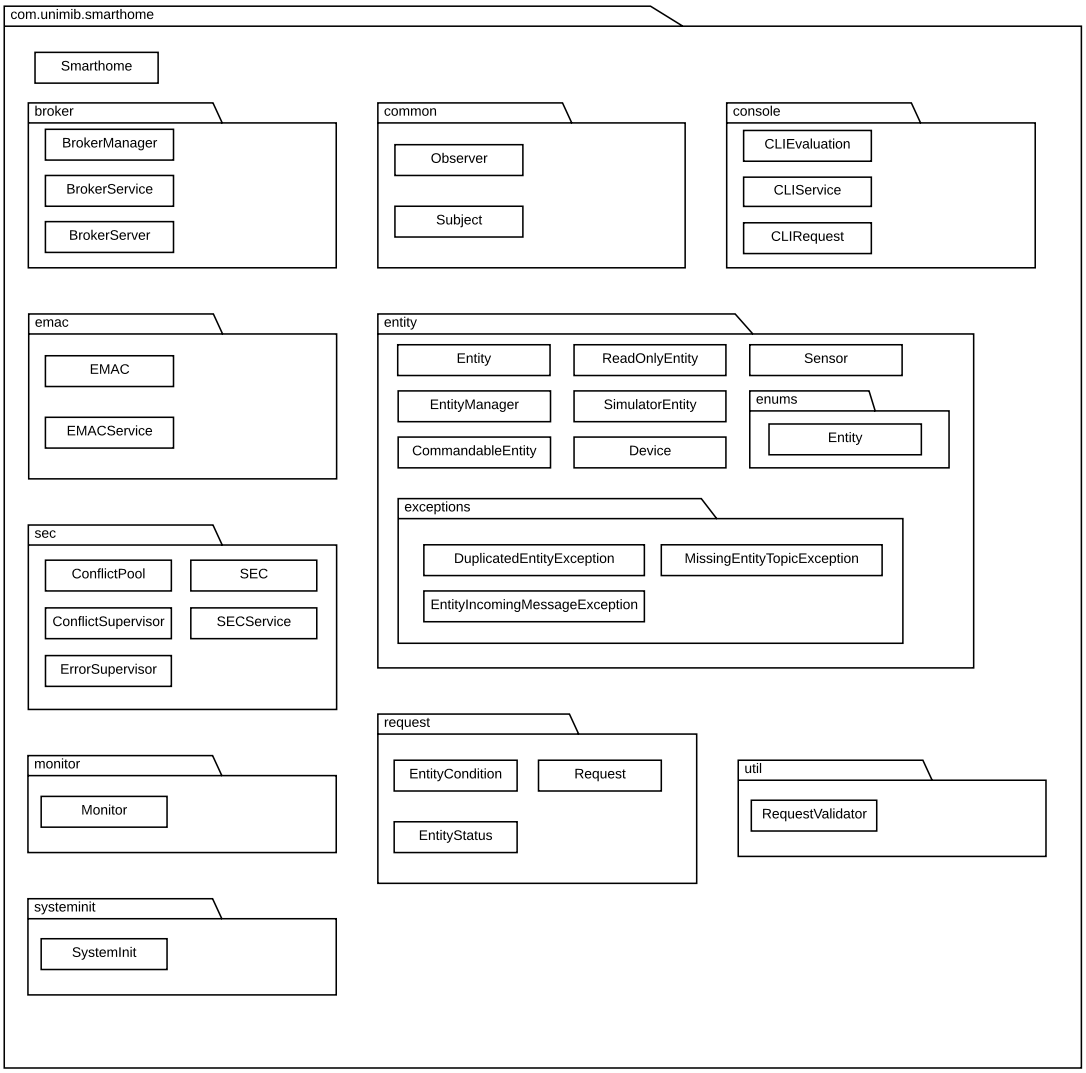
2.2.2 Attività del SEC



2.3 Diagramma dell'architettura software



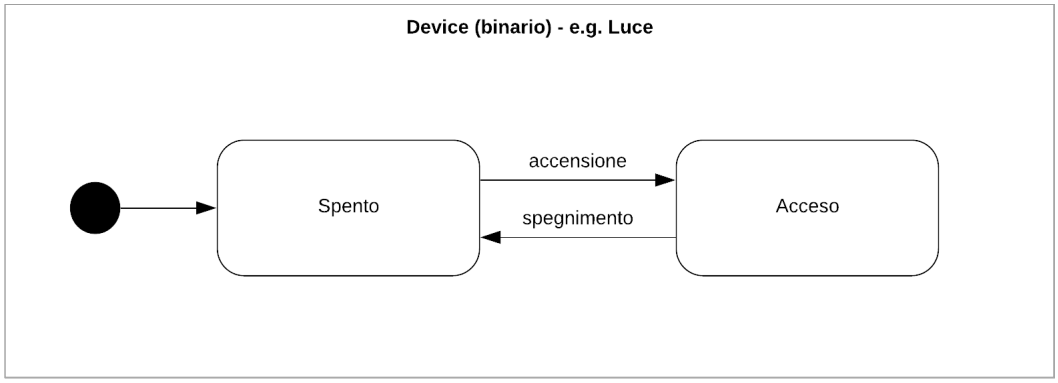
2.3.1 Diagramma dei package



2.4 Diagramma degli stati

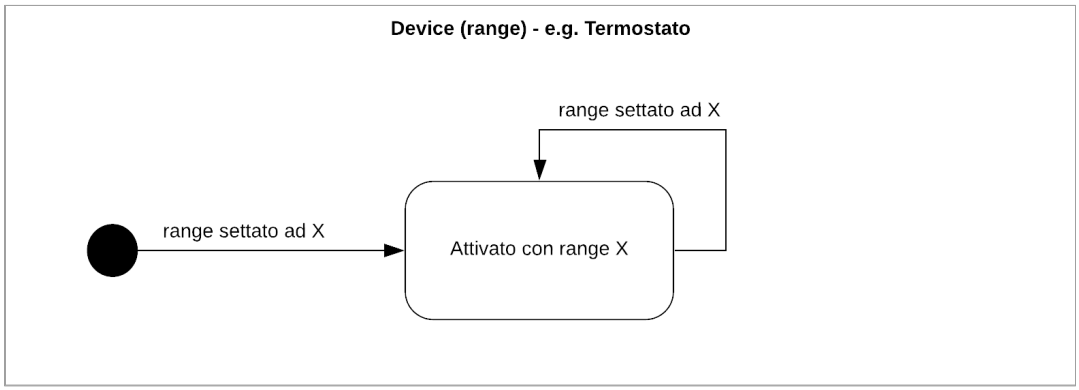
2.4.1 Device (binario)

I device binari (con entityType = BINARY) sono quei device che presentano due soli stati: spento ed acceso:



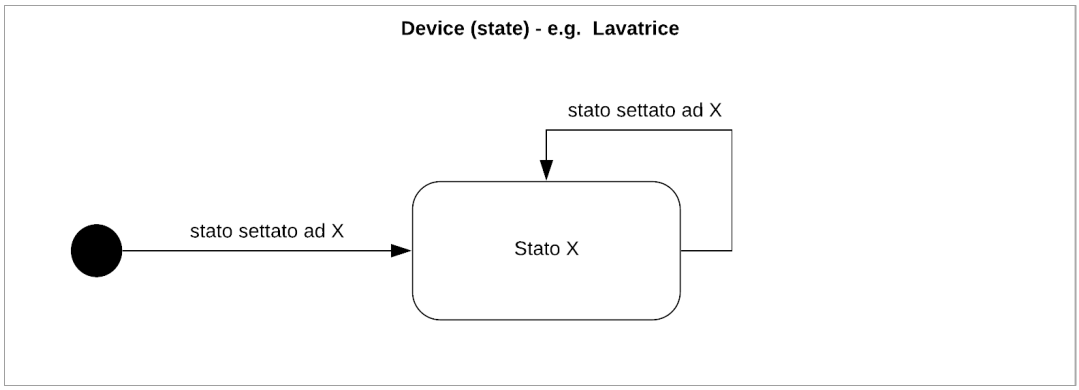
2.4.2 Device (range)

I device range (con entityType = RANGE) sono quei device che presentano un range di valori entro i quali possono essere impostati:



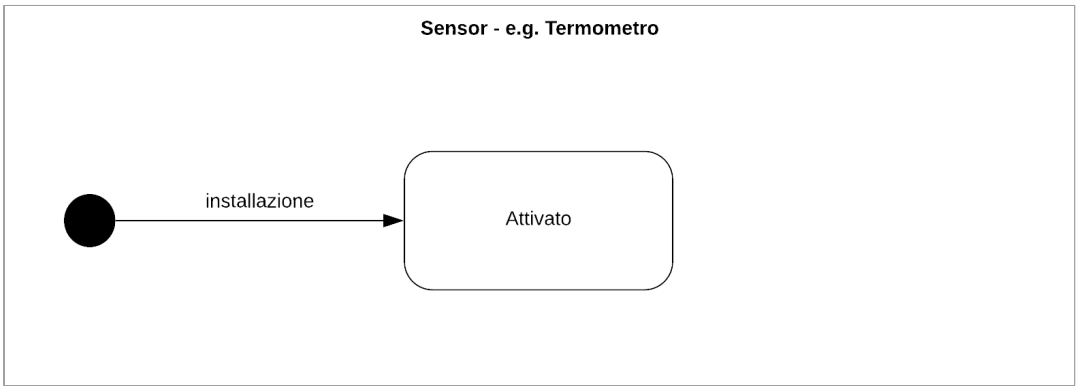
2.4.3 Device (state)

I device state (con entityType = STATE) sono quei device che presentano una serie di stati in cui possono trovarsi:



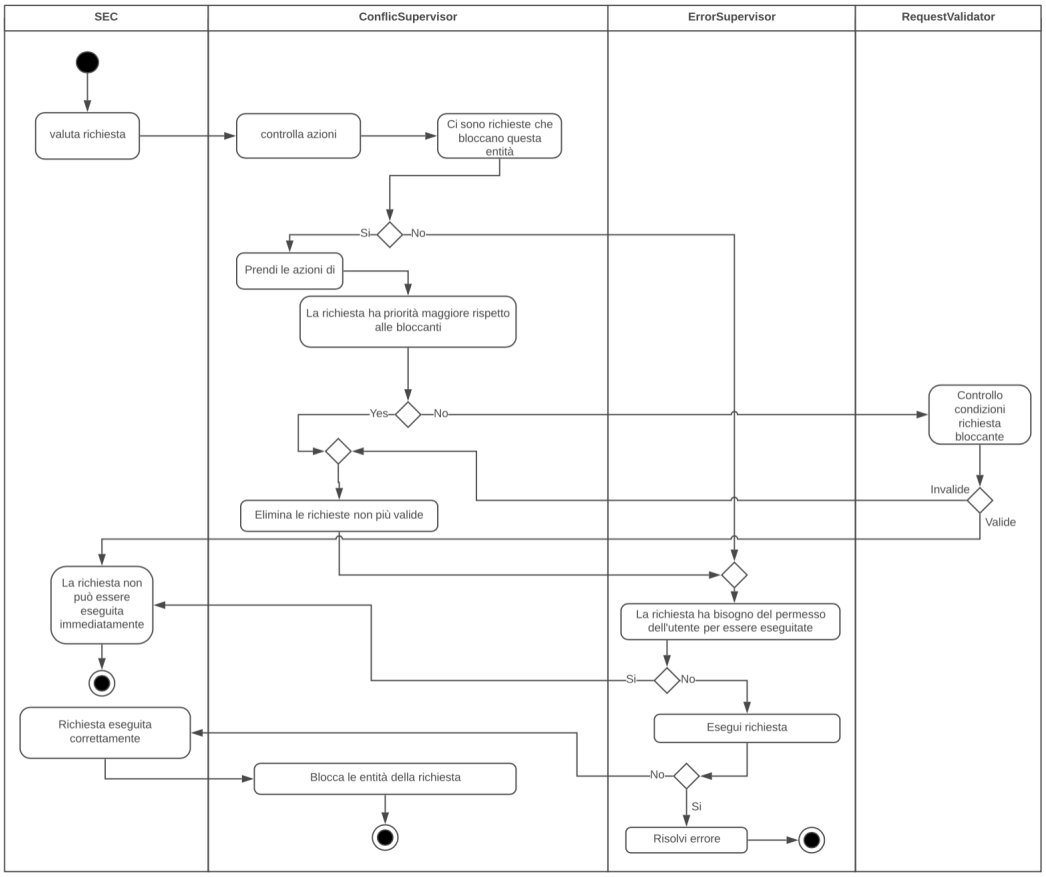
2.4.4 Sensor

I sensori sono quelle entità che riportano un valore che misurano. Nel nostro programma sono considerati solo dall'accensione in poi, in quanto spenti non avrebbero alcun significato:

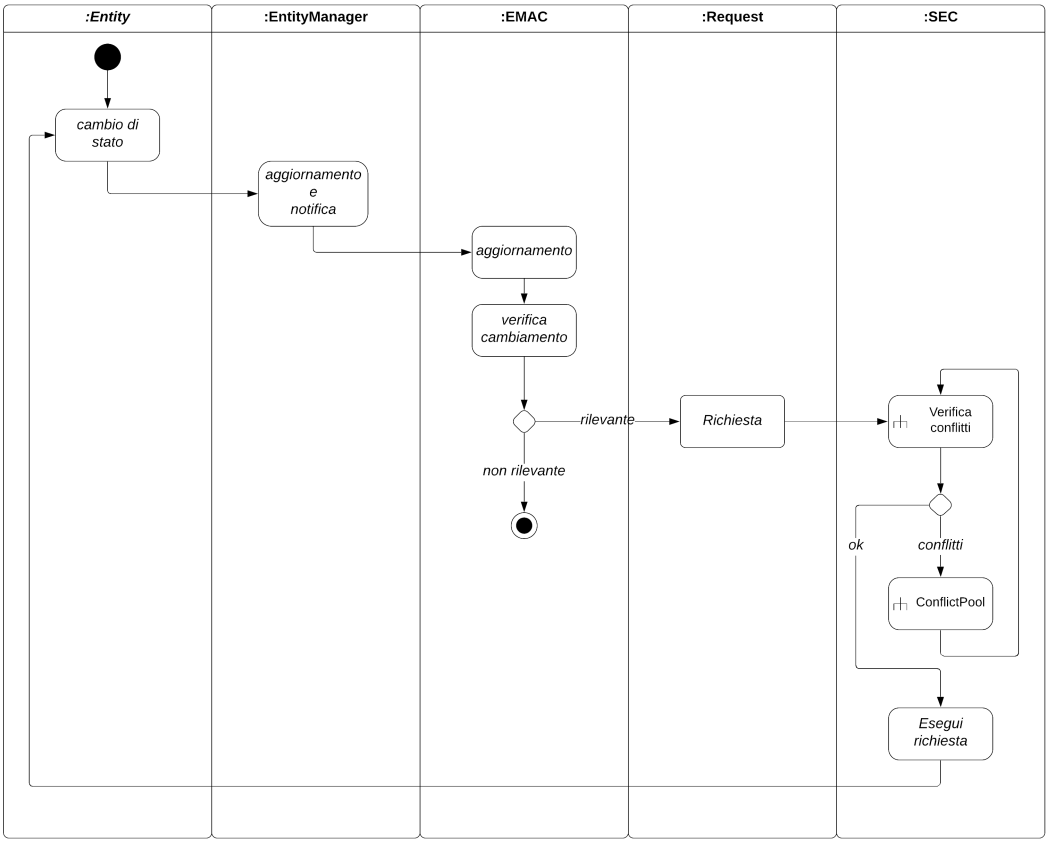


2.5 Diagramma delle attività

2.5.1 SEC



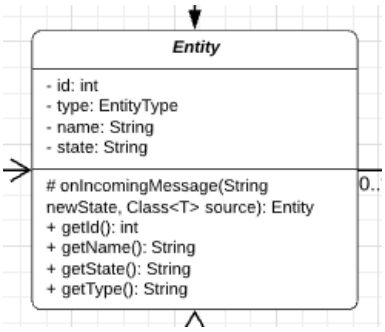
2.5.2 EMAC


:Entity
:EntityManager
:EMAC
:Request
:SEC
cambio di stato
aggiornamento e notifica
aggiornamento
verifica cambiamento
rilevante
non rilevante
Richiesta
Verifica conflitti
ok
conflitti
ConflictPool
Esegui richiesta

3. Entita

3.1 Astrazione

L'astrazione della classe entità è pensata in modo da permettere la creazione di **qualsiasi** tipo di oggetto presente nell'ambiente così da garantire al sistema SmartHome una compatibilità assoluta con ogni dispositivo esistente.



Di ciascuna entità si tiene traccia di un *identificatore*, di un *nome*, di un *tipo* e infine di uno *stato*:

- L'identificatore (*id*) serve a tenere un riferimento dell'entità all'interno del sistema.
- Il nome (*name*) invece serve solo per una corretta comunicazione con l'utente di tutti gli eventi del sistema che riguardano quella determinata entità.
- Il tipo (*type*) serve a definire il tipo dell'entità all'interno del sistema, così da poter essere coerenti con i controlli e le impostazioni da applicare quando una determinata entità è coinvolta.
- Lo stato (*state*) serve a tenere traccia dello stato **in tempo reale** dell'entità all'interno del programma.

Il metodo *onIncomingMessage(String newState, Class<T> source)* è molto importante per la natura dell'entità in quanto definisce il comportamento dell'entità quando riceve un messaggio dall'esterno (*source*). Esso è responsabile dei controlli da effettuare sul messaggio in ingresso ed eventualmente lancia una *EntityIncomingMessageException* qualora rilevi qualche irregolarità (ad esempio uno stato invalido per quel tipo di entità). Quando si lancia una *EntityIncomingMessageException* si entra in una fase molto importante e delicata del sistema, in quanto esso si accorge di un errore durante l'esecuzione di una richiesta e in autonomia prende una decisione su cosa fare quando ciò avviene. Le decisioni automatiche prese quando un'entità lancia un errore non sono state modellate, in quanto esse sono strettamente dipendenti dall'integrazione del sistema con l'ambiente (può essere un errore sul protocollo, un errore di lettura, un falso positivo, dipende da fattori troppo specifici per essere generalizzati in un unico caso).

La classe Entity è una classe **astratta** e **immutabile**. Il metodo *onIncomingMessage(String newState, Class<T> source)* ritorna un oggetto di tipo Entity che rappresenta il nuovo stato dell'entità (che verrà automaticamente registrato dal gestore delle entità).

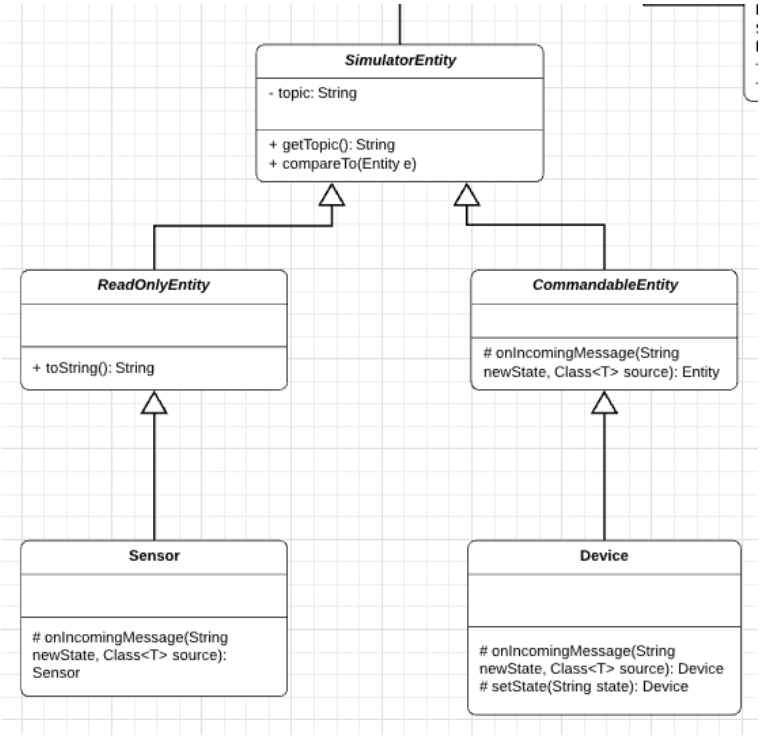
La classe Entity implementa anche l'interfaccia *Comparable<Entity>* per dare la possibilità ad una entità di poter essere correttamente comparata con un'altra (Lo stato può essere qualsiasi cosa).

Abbiamo scelto di usare String per rappresentare uno stato, in quanto grazie a questa implementazione è possibile rappresentare tutto (interi, gruppi di interi, caratteri ecc...).

3.2 Entità del Simulatore

Il simulatore di SmartHome per la comunicazione con il sistema utilizza un protocollo chiamato **MQTT** il cui funzionamento è basato sui topic. Ciascuna entità ha un topic, che si può interpretare come un canale, su cui viaggiano tutti i messaggi inerenti allo stato. Ad esempio: se l'entità Luce ha il topic "lights/light1" allora se si scrive 1 su quel topic (canale) sia il simulatore che il server sapranno che lo stato della luce è 1. La scrittura può essere fatta da chiunque sia sottoscritto al topic e il messaggio viene ricevuto da tutti i client sottoscritti.

Collegandoci a quando precedentemente detto, grazie all'astrazione implementata, ci risulta semplice creare una classe per astrarre tutte le entià del simulatore che, come informazione aggiuntiva, hanno un *topic*.



Successivamente la classe **SimulatorEntity** può essere distinta in **ReadOnlyEntity** e **CommandableEntity** la cui unica differenza sarà nell'implementazione di *onIncomingMessage(...)*. La classe *Sensor* che estende *ReadOnlyEntity* nell'implementazione di *onIncomingMessage(...)* controllerà che il messaggio sia proveniente dal simulatore (paramentro *source*) e solo in quel caso ne aggiornerà lo stato all'interno del sistema. La classe *device* che estende *CommandableEntity* non si porrà questo problema, in quanto generalizzando un dispositivo comandabile, potrà essere comandato dal sistema SmartHome.

La classi *SimulatorEntity*, *ReadOnlyEntity*, *CommandableEntity* sono classi **astratte**.

In SimulatorEntity lo stato sarà sempre rappresentato da un intero; perciò avendo sempre una stringa contenente un intero il metodo *compareTo(Entity e)* può essere correttamente implementato confrontando le due stringhe castate a interi.

3.3 Esempi di Entita

Per far comprendere le potenzialità del sistema vogliamo fornire un esempio di entità che possono essere tranquillamente implementante nel sistema senza preoccuparsi di eventuali errori derivanti da una diversa implementazione della gestione degli stati.

- **TimingEntity** se si volesse avere un clock interno da usare nelle automazioni, un'entità come TimingEntity potrebbe essere implementata mettendo nello stato un valore HH:mm che viene aggiornato ogni minuto.
- **RGBLightEntity** se si volesse implementare una luce RGB, basta settare come stato il colore esadecimale della luce "FFFFFF" e poi controllarla come le entità precedentemente viste

4. GetStarted

4.1 Installazione

4.1.1 SmartHome

Installare il sistema SmartHome non richiede step particolari diversi da quelli solitamente usati per l'installazione di un progetto Java:

1. Scaricare il progetto da GitHub
2. Aggiornare, se necessario, le dipendenze maven
3. Avviare il sistema dalla classe *SmartHome* dentro il package *com.unimib.smarthome*

Il sistema è stato sviluppato usando il **JDK 13**. (Bug 4.8.1))

Il sistema SmartHome supporta la connessione con un solo simulatore per volta.

4.1.2 SmartHomeSimulator

Per installare il Simulatore basta scaricare la versione compatibile da GitHub e avviare l'eseguibile .
Se dovessero esserci problemi con l'avvio del simulatore guardare la sezione Problemi 4.8 o Bug 4.8.1.

E' possibile avviare il simulatore anche dal progetto clonato da GitHub lanciando i seguenti comandi dalla root del progetto:

1. **cd HouseEntitySimulator**
2. **cd npm i** per installare tutte le dipendenze nel *package.json*
3. **cd npm start** che come specificato nel file *package.json* è uno script che lancia automaticamente il comando *electron .*

Queste istruzioni sono state testate con la versione **6.13.4** di NodeJS.

La connessione fra il sistema e il simulatore è automatica, nessuna azione è richiesta da parte dell'utente.

4.2 SmartHomeSimulator

SmartHomeSimulator è un software accessorio, a cui non abbiamo dedicato tempo per analisi e progettazione, scritto in JavaScript usando il framework Electron [1]

Figura 4.1: Frame di HouseEntitySimulator dove si possono notare i 4 tipi di entità: switch (es: luce2), che ha un solo un pulsante blu per cambiare lo stato; range (es: sensore gas), che con uno slider permette di settare il valore dentro uno specifico range; state (es: Allarme), che permette di selezionare da un menu a tendina lo stato; range-state (es: Tenda) che è un'unione nella stessa entità sia di range che di state

Il simulatore è in grado di simulare 4 tipi di entità:

1. **Switch**, cioè tutte le entità che possono avere come stato solo 0/1.
2. **Range**, cioè tutte le entità che possono avere come stato un valore dentro uno specifico range.
3. **State**, cioè tutte le entità che associano ad uno stato una stringa (es: 0 -> disattivato, 1 -> in chiamata). Non ci sono limiti sul valore dello stato, tutti gli interi sono utilizzabili.
4. **Range-State**, un'entità che è sia di tipo *Range* che di tipo *State*; avrà simultaneamente 2 stati: uno per il range e uno per lo state.

Non appena si avvia, il simulatore segnerà il proprio status di connessione con il sistema SmartHome nel box in alto a destra. Si consiglia di avviare prima il sistema e poi il simulatore.

Il simulatore prevede le seguenti shortcut da tastiera:

- **F12** per aprire i DeveloperTools e guardare la console .
- **CTRL + R** per ricaricare il Simulatore e riportalo allo stato iniziale. Lo stato iniziale sarà successivamente ricomunicato al sistema SmartHome.
- **ALT** per mostrare il menù di default di electron (Solo Windows).

Interagire con un'entità nel simulatore comporta ad una notifica istantanea al sistema SmartHome del nuovo valore impostato.

4.3 SmartHome

SmartHome è il sistema progettato per la gestione automatizzata delle entità, composto da vari Servizi che per semplicità sono stati implementati come Thread di SmartHome.

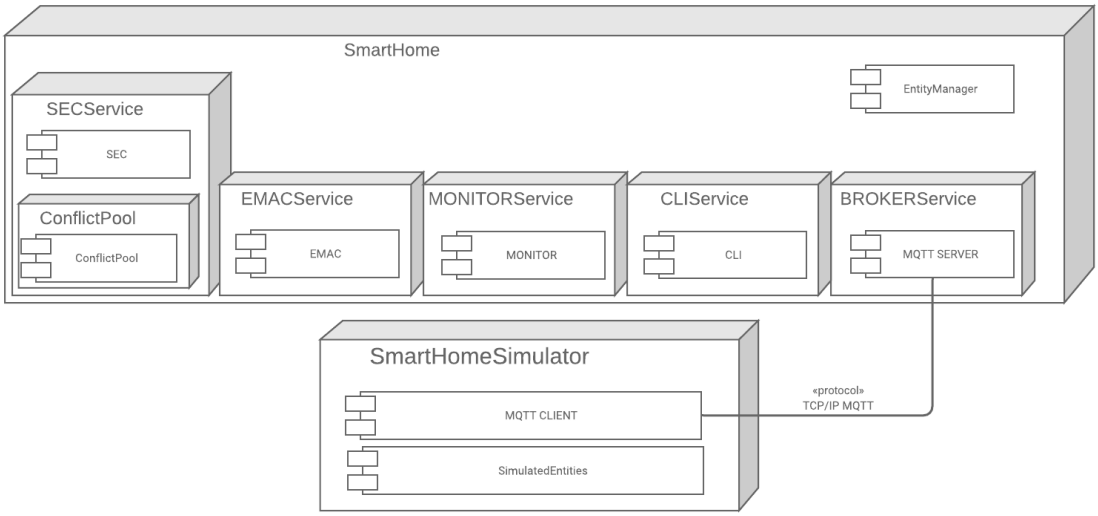


Figura 4.2: Rappresentazione dei Thread di SmartHome e della comunicazione con il simulatore

4.3.1 System for Errors and Conflicts

L'esecuzione del SEC e della Conflict Pool avvengono all'interno di **SECService**. Il SEC a sua volta coopera con 2 ulteriori classi per garantire la corretta esecuzione di una richiesta.

ConflictSupervisor

Questa classe ha lo scopo di rilevare possibili conflitti fra le varie richieste. Essa tiene traccia di tutte le richieste con il flag *retain* attivo (Sezione 4.4) mappando per ogni entità, una lista di richieste che la tengono bloccata.

Non appena si deve valutare una richiesta, basterà effettuare i controlli indicati nel diagramma delle attività 2.5.1 per garantire una corretta gestione dei conflitti con essa.

ErrorSupervisor

Con questa classe si rilevano i possibili errori generati dall'esecuzione di una richiesta. Per rilevare un errore la classe tenta di notificare l'entità chiamando il metodo *onIncomingMessage(...)* [3.1] e da questa chiamata capisce se il nuovo stato nell'entità genera un errore.

Se l'attributo *askPermission* 4.4 nella richiesta è attivo SEC informerà la CLI che è necessario chiedere il permesso all'utente; la richiesta verrà eseguita solo se le condizioni necessarie sussistono ancora nel momento in cui essa viene accettata.

Per la gestione automatica degli errori si tralascia l'implementazione in quanto essa è fortemente dipendente dall'ambiente esterno e per gestirli è necessario prevedere il tipo di errori che l'ambiente può generare.

Se una richiesta richiede un permesso specifico per essere eseguita, la classe *ErrorSupervisor* non esegue la richiesta ed informa il servizio della CLI 4.3.3 di tale necessità.

ConflictPool

Infine questa classe si occupa di raccogliere tutte le richieste che hanno generato un conflitto e prova a rieseguirle ogni 10s. Una richiesta per finire dentro la Conflict Pool è stata lanciata o dall'utente o dall'EMAC e perciò si immagina come un'azione da eseguire in ogni caso non appena possibile. Nella sezione dove si introducono le richieste sono forniti ulteriori esempi 4.4

4.3.2 Entity Monitor Automation Controller

L'EMAC è gestito all'interno di **EMACService**. Questa classe si occupa di analizzare tutti i cambiamenti di stato delle varie entità ed, eventualmente, di lanciare la richiesta prevista da quella automazione.

In questa classe avviene il controllo delle condizioni di un'automazione. 2.2.1

4.3.3 Command Line Interface

La CLI è gestita all'interno di **CLIService**. Questo servizio offre un'interazione da linea di comando con il sistema SmartHome. i comandi disponibili sono:

1. **list** per ottenere la lista di tutte le informazioni delle entità presenti nel sistema
2. **set entityID status retain priority** questo comando da la possibilità di settare lo stato di un'entità. Se non specificati i valori *retain* e *priority* hanno di default i valori *false* e *MAX-INT*.
3. **get entityID** per ottenere le informazioni su una specifica entità.
4. **clearCP** per cancellare la Conflict Pool.
5. **listCP** per cancellare la Conflict Pool.
6. **accept** per accettare una richiesta in attesa di autorizzazione da parte dell'utente.
7. **refuse** per rifiutare una richiesta in attesa di autorizzazione da parte dell'utente.
8. **shutdown** per spegnere il sistema SmartHome.

4.3.4 Monitor

La classe *Monitor* gestita da **MonitorService** scrive su un file tutti i cambiamenti di stato.

Essa si occuperà anche di effettuare analisi sui dati raccolta qualora ve ne sia la necessità; l'implementazione dell'analisi è stata tralasciata.

4.3.5 Broker

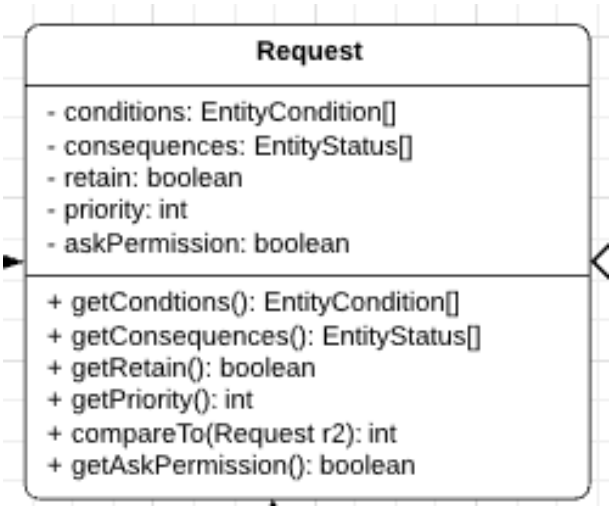
BrokerService è il servizio che si occupa di intercettare tutte le richieste del simulatore che viaggiano attraverso il protocollo Message Queue Telemetry Transport (MQTT).

Non appena rileva una nuova connessione in ingresso chiede un aggiornamento sugli stati di tutte le entità per sincronizzarsi con i dati visualizzati dal simulatore.

La classe tiene traccia di una sola connessione alla volta sovrascrivendo ogni nuova connessione in ingresso con quella precedente.

4.4 Request

Una richiesta nel diagramma di progetto 2.1.3 è stata definita nel seguente modo



Lo scopo principale di questa implementazione è quello di permettere ad una richiesta di fare potenzialmente qualsiasi cosa voglia all'interno del sistema.

l'attributo *conditions* definisce l'insieme di condizioni da verificare prima di effettuare un'azione.

Tali verifiche possono essere effettuate sia dall'EMAC, quando riceve una notifica di cambiamento, sia dal SEC quando deve verificare se le condizioni di una richiesta sono ancora valide.

Quando il SEC esegue una richiesta, esegue tutte le azioni specificate nell'attributo *consequences*.

Il flag *retain* serve a bloccare un'entità su quella richiesta. Tutte le richieste con priorità minore a quella che blocca l'entità verranno scartate e inserite nella Conflict Pool. L'unico modo per **rimuovere il blocco** da un'entità è quello di eseguire una richiesta con l'attributo *priority* maggiore rispetto a quello della richiesta bloccante.

L'attributo *askPermission* serve per specificare quali azioni debbano avere il permesso dell'utente per essere eseguite.

4.5 File di Configurazione

4.5.1 SmartHome

Nella nostra implementazione di SmartHome tutte le entità e tutte le automazioni presenti nel sistema sono caricate da un file json.

I files di configurazione si trovano nella cartella *resources*.

Entità

Per informare il sistema SmartHome di quali entità deve controllare (del simulatore) bisogna modificare il file *entities.json* impostato nel seguente modo:

```
{
   "entities":[
      {
         "name": "Sensore temperatura",
         "topic": "sensor/temp1",
         "type": "RANGE",
         "id": 1,
         "commandable": false
      },
      ...
    ]
}
```

entities è un array di entità. Ogni entità deve avere presente i parametri:

- ***name***: il nome dell'entità nel sistema.
- ***topic***: il topic dell'entità con cui andrà a comunicare col simulatore. I topic sono **casesensitive**.
- ***type***: il tipo di entità (RANGE, STATE, BINARY).
- ***id***: l'id dell'entità **univoco** nel sistema.
- ***commandable***: se l'entità è comandabile (istanzia da *Device*) oppure no (istanzia da *Sensor*).

Automazioni

Per informare il sistema SmartHome di quali automazioni si deve occupare bisogna modificare il file *automations.json* impostato nel seguente modo:

```
{
   "automations":[
      {
         "conditions": [
                                 {
                                          "id": 4,
                                          "value": 0,
                                          "rel": "="
                                 },
                                 ...
                        ],
                        "consequences": [
                                 {
                                          "id": 5,
                                          "value": 0
                                 },
                                 ...
                        ],
                        "retain": false,
                        "priority": 1,
                        "ask": true
      },
      ...
   ]
}
```

Ogni **Automazione** deve avere i parametri:

- ***conditions***: Array di *condizioni*, l'automazione per essere verificata deve avere tutte le condizioni **valide**.
- ***consequences***: Array di *conseguenze*, esse sono tutti gli aggiornamenti di stato da apportare alle varie entità se le condizioni sono verificate.
- ***retain***: booleano che indica se la richiesta deve essere bloccante o no (per maggiori dettagli 4.4).
- ***priority***: intero che indica il livello di priorità della richiesta.
- ***ask***: booleano che indica se la richiesta necessita o no dell'autorizzazione dell'utente.

Ogni **condizione** deve avere i parametri:

- ***id***: id dell'entità da controllare.
- ***value***: valore che l'entità dovrebbe controllare.
- ***rel***: relazione fra valore attuale e valore da controllare ($=$, $>$, $<$).

Ogni **conseguenza** deve avere i parametri:

- ***id***: id dell'entità da aggiornare.
- ***value***: nuovo valore dell'entità da impostare.

LOG

Nel file di log è possibile impostare cosa mostrare in console

Aggiornando il valore *level* nella linea 30 del file *log4j2.xml*

```
<AppenderRef ref="Console" level="INFO"/>
```

le impostazioni consigliate sono:

- **INFO** per mostrare i messaggi da tutti i servizi (Eccetto monitor).
- **CLI** per mostrare i soli comandi dal servizio che gestisce la linea di comando.

4.5.2 SmartHomeSimulator

Il simulatore, come il sistema SmartHome, carica le impostazioni delle entità da simulare da un file json chiamato *entities.json* (**N.B** sono 2 file diversi, questo si trova nella root del simulatore)

```
{
   "entities":[
      ...
    ]
}
```

Il parametro *entities* è un array di entità. Il simulatore supporta 4 tipo di entità, vengono di seguito evidenziati i template

SWITCH

```
{
  "name": "Luce",
  "switch_topic": "lights/light1",
  "type": "switch"
}
```

RANGE

```
{
  "name": "Sensore qualità aria",
  "range_topic": "sensor/air1",
  "type": "range",
  "range_min_val": 0,
  "range_max_val": 100
}
```

STATE

```
{
  "name": "Robot pulizia",
  "state_topic": "robots/cleaner1",
  "type": "state",
  "states" : [
    {
      "value": 0,
      "name": "Spento"
    },
    ...
  ]
}
```

RANGE-STATE

```
{
  "name": "Serratura automatizzata",
  "state_topic": "engine/door1/state",
  "range_topic": "engine/door1/position",
  "type": "range_state",
  "range_min_val": 0,
  "range_max_val": 100,
  "states": [
    {
      "value": 0,
      "name": "Chiuso"
    },
    ...
  ]
}
```

Si faccia riferimento alla sezione SmartHomeSimulator [4.2] per la differenza fra queste.

N.B i nomi dei topic cambiano fra i vari tipi di entità.

I valori iniziali delle entità simulate sono:

- *range-min-val* per le entità di tipo *RANGE*
- il *primo* statno dell'array *states* per le entità di tipo *STATE*
- *1* per le entià di tipo *SWITCH* (guarda il bug 4.8.1)

4.6 Configurazione Iniziale

4.6.1 Automazioni

Nel nostro sistema sono presenti diverse automazioni:

- Nel caso in cui il sensore *conta persone 1* abbia impostato come stato 0 e la luce "Luce sala" sia accesa, il sistema genera un messaggio in console, dove chiede se si vuole procedere alla richiesta di spegnimento della luce, bisogna rispondere "accept" oppure "refuse" all'interno della console.
- Nel caso in cui il sensore *conta persone 2* non rilevi nessuna persona all'interno della stanza, spegnerà in automatico la luce "Luce camera" nel caso in cui questa risulti accesa.
- Quando il sensore di gas rileva una percentuale di gas troppo elevata all'interno della stanza(10%) il sistema apre le finestre finché la percentuale non diminuisce; inoltre se l'allarme risulta attivo, questo viene disattivato per evitare di effettuare una chiamata d'emergenza.
- Nel caso in cui l'allarme sia impostato su "attivo totalmente" e il sensore di movimento risulti attivo, il sistema deve spegnere tutte le luci, chiudere tutte le serrature e fare una chiamata di emergenza; questa automazione viene eseguita anche se la serratura risulta aperta.
- Se l'allarme risulta impostato su "attivo esterno" significa che deve effettuare una chiamata di emergenza solo nel caso in cui le serrature risultino aperte.
- Quando il sensore di temperatura rileva una temperatura sotto un certo livello(24) oppure è troppo elevata(26) il nostro sistema deve attivare il termostato, così da riportare la temperatura fra i 23 e i 26 gradi dove il termostato risulterà spento.
-

4.7 Debug

Il sistema SmartHome è stato progettato per fornire informazioni testuali sulle principali azioni intraprese da ogni servizio.

I log dell'applicazione sono disponibili all'interno del file *smarthome.log*.
Se si volessero visualizzare in console tutte le informazioni utili al debug si guardi come attivare questa configurazione nella sezione [4.5.1].

Per visualizzare i log sul simulatore basta aprire i DeveloperTools (F12) e andare nella tab console.

4.8 FAQ

In questa sezione sono raccolti i problemi più frequenti che si possono verificare.

Perchè nel simulatore non ci sono entità
: Si controlli il file *entities.json* che deve essere nella root del simulatore. Se il file non è stato cancellato/spostato probabilmente si tratterà di un errore di sintassi, si guardi la console del simulatore per capire dove sta il problema.

Il simulatore non parte
: Si provi ad eseguirlo direttamente con node.

Il sistema SmartHome non riceve gli eventi del simulatore
: Si effettui il debug di entrambi i sistemi; se nella console del simulatore non ci sono errori allora si verifichi che nei log di SmartHome il server riesca ad avviarsi correttamente nella porta 1883. Se il server riesce ad avviarsi ma non riceve una connessione in ingresso si provi a ricaricare il simulatore (anche CTRL+R va bene). Se nonostante il ricaricamento non funziona, probabilmente si tratta di un problema di connessione magari con qualche firewall che blocca le connessioni in locale.

Il sistema SmartHome non carica entità o le automazioni
: Si guardino i messaggi di log del sistema per possibili errori nei file json.

Il broker da l'errore MQTT server error on start Address already in use: bind
: Verificare che non ci siano processi aperti sulla porta 1883.

Se si dovessero verificare ulteriori problemi potete contattarci su Telegram.

4.8.1 BUG NOTI

- Quando il simulatore si collega a SmartHome inverte tutti gli stati delle entità di tipo switch (Per questo lo stato iniziale delle luci è 1 e si consiglia di avviare prima SmartHome e poi il simulatore) FIX: Avviare prima il sistema SmartHome e poi avviare il simulatore.
- Quando si spegne il sistema SmartHome il simulatore va in stato "riconnessione". Se si riaccende il sistema SmartHome il simulatore segnala "connesso" ma non riesce a comunicare con SmartHome. FIX: (CTRL+R) per ricaricare il simulatore. Tutte le entità verranno riportate allo stato iniziale.

5. Commenti

5.1 Design Principles

5.1.1 Single Responsibility

Abbiamo pensato la struttura del nostro software in modo tale che ogni classe avesse il suo specifico compito, così da semplificare al massimo le ineluttabili operazioni di refactoring del codice in fase di implementazione. Inoltre ciò ha permesso di evitare la creazione di una classe troppo grande, di cui sarebbe stato difficile tenere traccia in corso di sviluppo, anche solo per ricordare in dettaglio cosa facesse e, soprattutto, come lo facesse.

5.1.2 Liskov Substitution

Nella nostra modellazione tutte le sottoclassi possono essere utilizzate in luogo della propria superclasse senza causare errori o conflitti. L'overriding dei metodi non ne stravolge la natura, ma la specifica adattandola alle necessità della sottoclasse.

5.1.3 Open/Closure

Il nostro progetto è aperto per le estensioni in virtù della sua forte struttura gerarchica (si veda la voce "Hierarchy"): sarà possibile in futuro estendere le funzionalità delle sue classi sia dal punto di vista qualitativo che quantitativo (i.e. sia migliorare cosa possono fare, sia aumentare il numero di scenari in cui lo possono fare).

5.1.4 Precisazione prima di continuare

Ora procederemo ad analizzare l'utilizzo dei design principle di Sharma (i cosiddetti PHAME): si tenga presente che questi principi trovano parecchi punti in comune con i principi SOLID appena trattati, per cui la lettura potrebbe risultare ridondante. Nonostante ciò desideriamo mantenere una struttura a sezioni per facilitare la lettura rapida (con il feel di un manuale, per intendersi).

5.1.5 Hierarchy

Il nostro progetto presenta una forte struttura gerarchica, che si avvale dell'uso di generalizzazioni piene di significato e ordinate secondo il grado di specificità che intendono rappresentare: ogni classe ha il suo particolare utilizzo, ma può comunque essere sostituita in vece di una sua classe padre (principio di Liskov Substitution).

5.1.6 Abstraction

Come già accennato, è presente un grande utilizzo dell'astrazione, sempre accompagnata da motivazioni quali la ricerca di diversi gradi di specificità e la volontà di evitare duplicati all'interno del codice. Così facendo siamo riusciti a sopperire ai diversi specifici bisogni che intendevamo soddisfare tramite il nostro sistema.

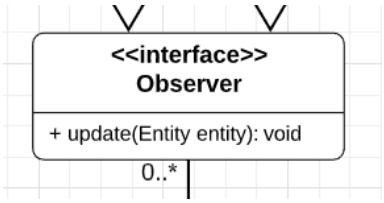
5.1.7 Encapsulation

Dove possibile, abbiamo cercato sempre di mantenere al minimo la fuoriuscita di informazioni dallo scope della classe che le conteneva: ad esempio l'utilizzo di un id per le Entity permette di non passare direttamente l'oggetto (con tutte le informazioni che contiene) "in giro" per il codice, ma di recuperare successivamente solo le informazioni necessarie tramite i metodi get.

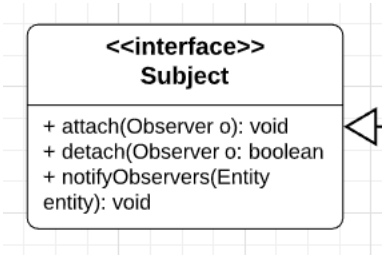
5.2 Design Patterns

5.2.1 Observer

Il design pattern Observer è stato utilizzato per le classi EMAC e MonitorService, che necessitano di tenere traccia dei cambiamenti delle Entity in EntityManager rispettivamente per valutarli e registrarli. In particolare EMAC e MonitorService implementano l'interfaccia Observer, mentre EntityManager implementa Subject:



Observer presenta il metodo update(Entity entity), che si occupa di aggiornare lo stato delle Entity osservate dall'Observer (EMAC o MonitorService).



Subject presenta i metodi attach(Observer o) e detach(Observer o), che servono rispettivamente per aggiungere o rimuovere un Observer dallo scope di un Subject; notifyObservers(Entity entity) chiama l'update(Entity entity) su tutti gli Observer che sono salvati nell'oggetto che implementa Subject.

5.2.2 Singleton

EMAC, EntityManager, SEC e MonitorService sono classi Singleton, vale a dire che non possono essere generati chiamando il loro costruttore, hanno una sola istanza globale condivisa da tutto il programma (static), e presentano un metodo getInstance() che la restituisce e ne garantisce l'unicità.

Initialization on Demand

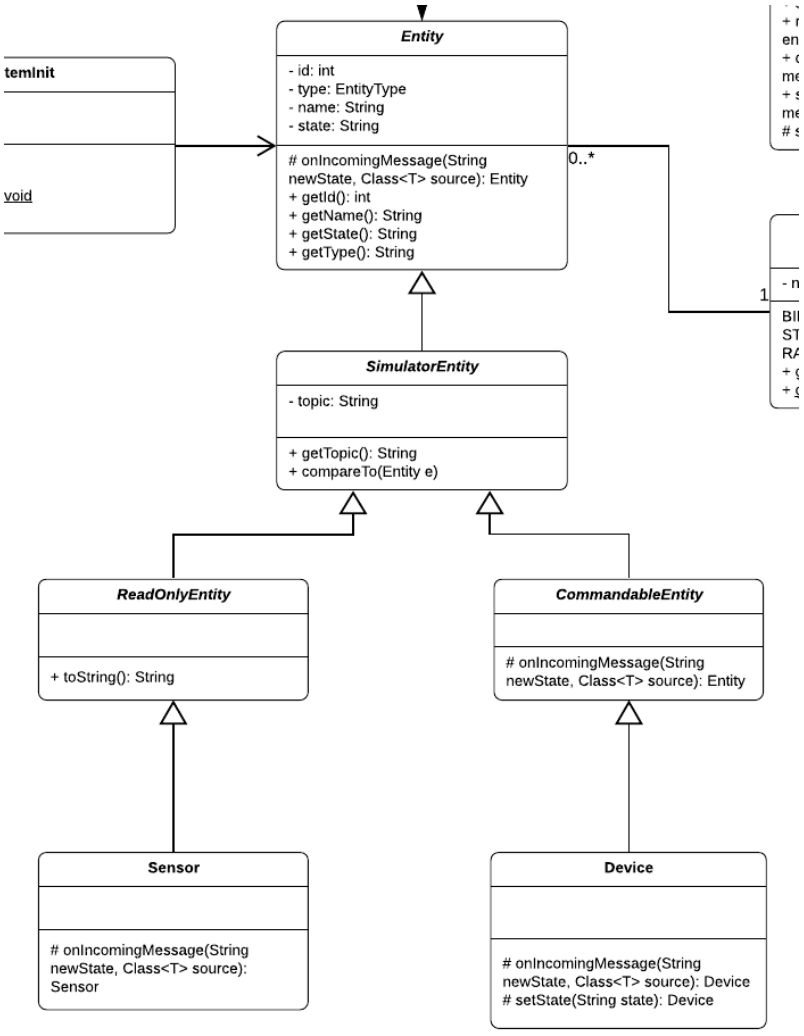
In particolare la nostra implementazione di Singleton include il design pattern Initilization on Demand, che fa uso di una classe statica LazyHolder per il lazy loading delle istanze delle classi singleton. LazyHolder è inclusa in tutte le classi singleton ed è proprio essa a contenere il metodo getInstance().

```
//...
private static class LazyHolder {
                private static final EMAC INSTANCE = new EMAC();
        }

        public static EMAC getInstance() {
                return LazyHolder.INSTANCE;
        }
//...
```

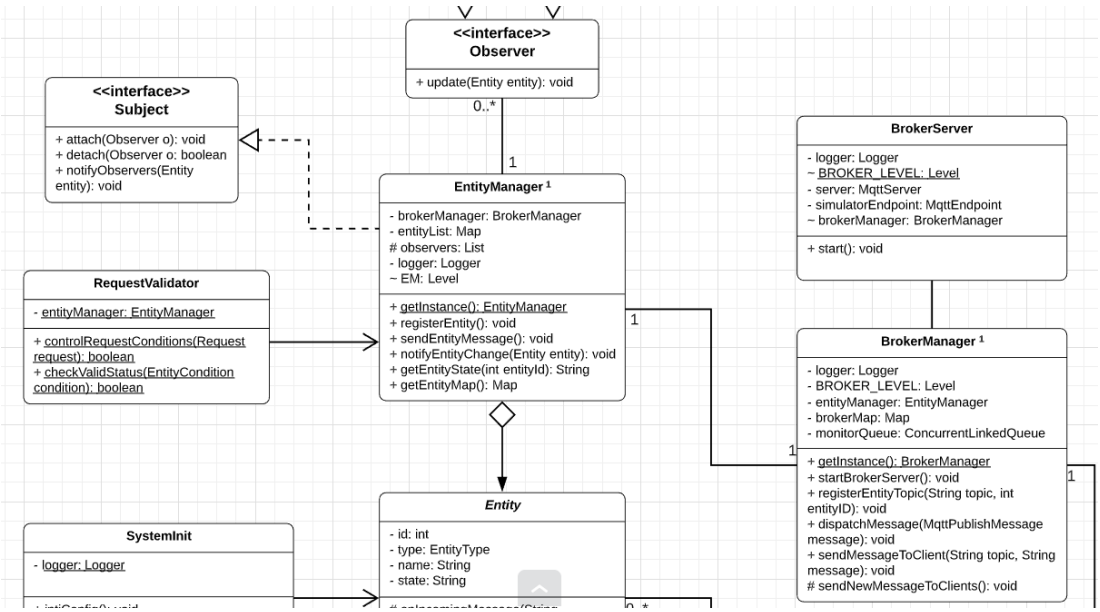
5.2.3 Template Method

"Il template method è un design pattern comportamentale che definisce lo scheletro di un algoritmo nella superclasse ma lascia che le sottoclassi facciano l'override di specifici passi dell'algoritmo senza cambiare la sua struttura" [2]. E' esattamente ciò che accade col metodo OnIncomingMessage(String newState, Class<T> source) di Entity:



5.2.4 Mediator

Nonostante la struttura intrinseca del pattern non sia ripresentata alla lettera (non abbiamo un'interfaccia Mediator), l'idea di utilizzare un oggetto che si occupi della comunicazione tra altri oggetti è completamente abbracciata dalla nostra implementazione. Infatti EntityManager non è altro che un oggetto intermediario tra le Entity e gli altri: non è possibile accedere alle Entity se non tramite esso e le Entity non possono comunicare se non tramite esso. Mentre Observer si occupa solo della notifica dei cambiamenti di stato agli osservatori, Mediator permette di applicare le modifiche necessarie alle Entity e la comunicazione fra le classi necessarie per farlo:



5.3 Architectural Patterns

5.3.1 Monitor Object

Nel nostro progetto diverse classi sono accedute globalmente dal resto del programma (EMAC, SEC, EntityManager, BrokerManager): questo causa degli inevitabili problemi di concorrenza, ai quali si può far fronte con l'utilizzo di metodi synchronized per appunto sincronizzare le operazioni e soprattutto le possibili modifiche all'oggetto acceduto. Ed è proprio in questo che consiste il pattern Monitor Object: l'utilizzo di metodi sincronizzati per l'accesso concorrente garantisce l'assenza di "situazioni indesiderate" o modifiche di cui non possiamo gli effetti.

5.3.2 Domain Model

Il pattern Domain Model prevede che l'architettura del software abbia una struttura dei ruoli ben definita, con una suddivisione dei compiti tra le classi chiara e coerente. Ogni classe del nostro progetto infatti assolve ad una determinata responsabilità, interfacciandosi con le altre per comunicare i risultati delle proprie operazioni.

5.4 Analisi di Understand

Un'analisi con il tool Understand è stata effettuata sul codice per evidenziare eventuali antipattern strutturali.

5.4.1 Riepilogo del codice

Understand, in un'analisi sommaria, ha rilevato se seguenti misurazioni:

- Blank Lines 430
- Classes 39
- Code Lines 1.277
- Comment Lines 86
- Comment to Code Ratio 0,07
- Declarative Statements 538
- Executable Statements 392
- Files 34
- Functions 149
- Lines 1.773

5.4.2 Complessità ciclomatica

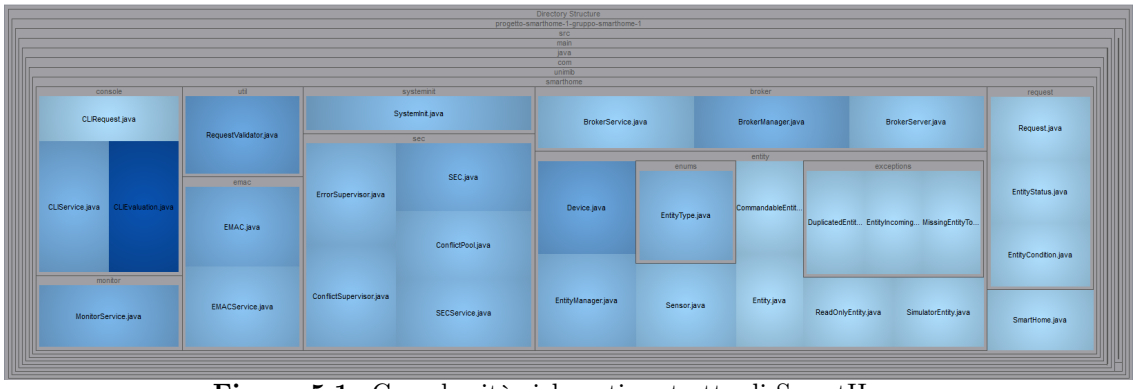


Figura 5.1: Complessità ciclomatica stretta di SmartHome

La Figura 5.1 mostra una rappresentazione del programma dove la mappatura dei colori è basata sulla complessità ciclomatica stretta. Da tale misurazione si evince che la classe con un valore ciclomatico "critico" è *CLIEvaluation* ed è pari ad 11. Questo accade perchè dentro la classe *CLIEvaluation* c'è uno statement *switch* per identificare il comando letto dalla console; se si effettua una misurazione per calcolare la complessità ciclomatica modificata, il valore "critico" scende da 11 a 4 perchè tutti gli statement *switch* vengono contati come un unico nodo.

Tutti i valori analizzati sono in un range 1-10 che secondo il Software Engineering Institute non comportano molti rischi nella gestione del programma.

5.4.3 Lavoro delle classi

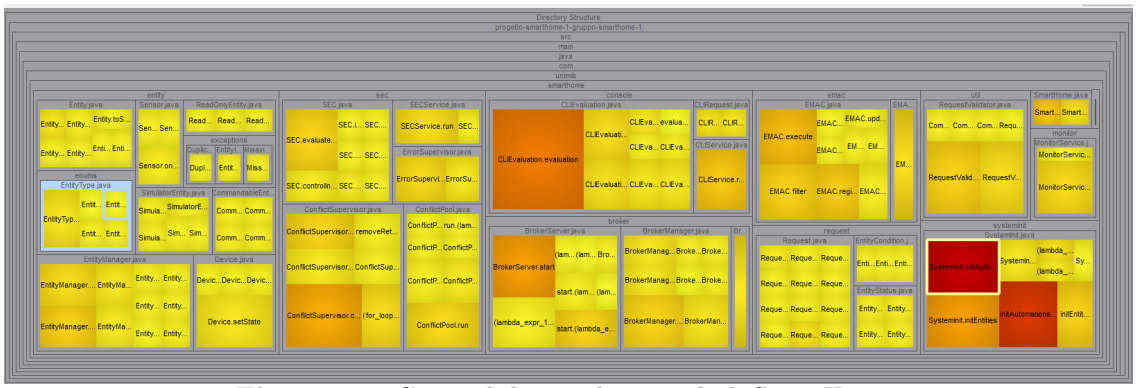


Figura 5.2: Carico di lavoro dei metodi di SmartHome

La Figura 5.2 mostra una rappresentazione del carico di lavoro dei metodi di SmartHome. Da questa misurazione è facile identificare se nel programma sono presenti vari codemells come *long-Method* o *featureEnvy*. I metodi più critici nel sistema sono *initAutomations* il cui carico di lavoro è effettivamente eccessivo. Abbiamo consapevolmente caricato questo metodo perché è necessario per l'inserimento delle automazioni all'interno del nostro sistema; questo avviene solo in fase di inizializzazione del sistema e non essendo un'operazione frequente riteniamo che non sia una priorità risolverla.

Il carico di lavoro dei metodi del programma, eccetto per i metodi dell'inizializzazione, è ben distribuito.

5.4.4 Violazioni

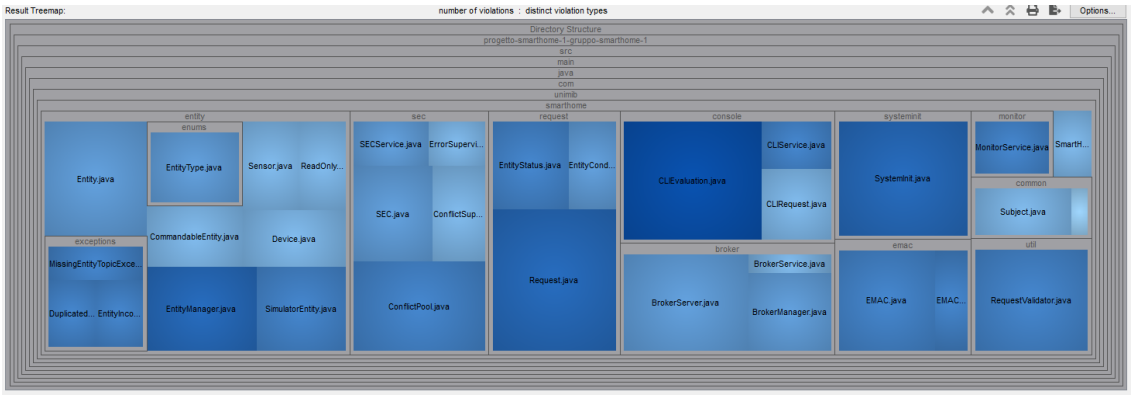


Figura 5.3: Violazioni nelle classi di SmartHome

La Figura 5.3 mostra una rappresentazione di tutte le violazioni presenti nelle classi di SmartHome. Per la maggior parte si tratta di violazioni di tipo *MagicNumber* (numeri senza significato per essere interpretati, vorrebbero una costante che ne spieghi il valore), *Program Unit Comment to Code Ratio* in quanto il codice non ha commenti a sufficenza e emphSingle exit point on method. Teniamo a precisare che le violazioni rilevate sono meno significative rispetto ai controlli sulle violazioni previste da SciTools di cui non ci sono rilevazioni con il codice di SmartHome.

5.4.5 Coupling

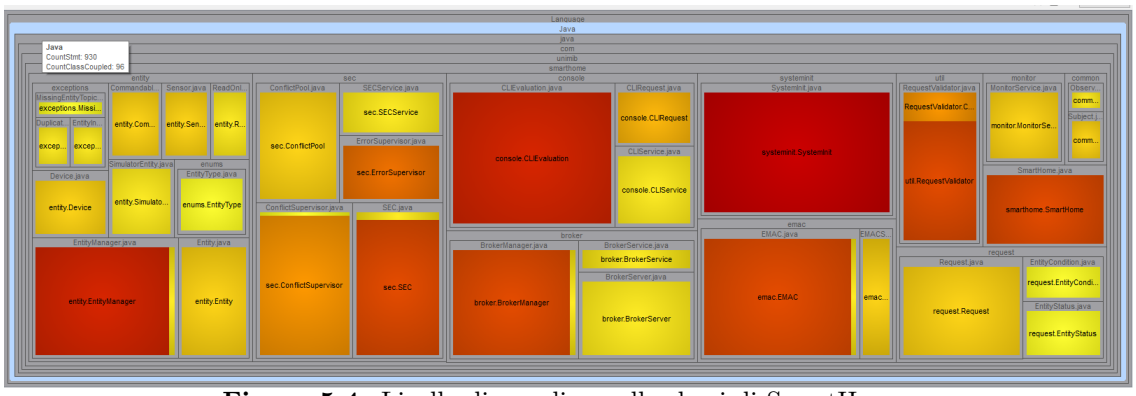


Figura 5.4: Livello di coupling nelle classi di SmartHome

La Figura 5.3 mostra una rappresentazione del livello di coupling fra le classi del sistema SmartHome. SmartHome è stata pensata come un'applicazione a microservizi, qundi ogni riferimento ad un a entità all'interno di un servizio dovrebbe passare attraverso l'istanza del servizio. Per non complicare ulteriormente il sistema abbiamo preferito creare un sistema con un'architettura a microservizi ma con un'implementazione monolitica. L'alto livello di accoppiamento nelle classi **core** del progetto (EntityManager, SEC, EMAC, ..) è giustificato da questa nostra scelta. Se si dovesse riadattare il sistema ad una vera e propria applicazione microservizi, grazie alla nostra progettazione e implementazione il riadattamento sarebbe semplice e si risolverebbero i problemi del coupling.

5.5 Analisi di SonarQube

L'analisi di SonarQube per il nostro codice riporta come risultato un grado A. In particolare il codice presenta:

- Zero bug: il sistema funziona e supera tutti i test
- Zero vulnerabilità;
- Un security hotspot: la risoluzione di questa problematica prevedeva l'utilizzo di un framework aggiuntivo che abbiamo preferito non implementare per evitare un ulteriore complicazione nell'architettura del progetto.
- Zero code smell;
- 0 minuti di debt;
- 81.3% di copertura da parte dei 17 test creati.

6. Conclusioni

Tutti gli obiettivi prefissatici sono stati raggiunti: un appunto particolare va fatto per la classe MonitorService, per la quale ci eravamo auspicati di ottenere uno scope più grande, che andasse anche ad abbracciare forme di analisi statistica, stima delle preferenze dell'utente e via discorrendo. Queste funzionalità, sebbene non siano stato implementate, potrebbero molto facilmente (il progetto è aperto alle estensioni) costituire uno spunto di espansione particolarmente interessante.

Da un punto di vista più personale ci sentiamo di esprimere il nostro stupore per quanto riguarda l'esperienza diretta con lo sviluppo software. Le dinamiche di tipo gestionale e quelle di tipo implementativo hanno sicuramente arricchito il bagaglio culturale in materia di ingegneria del software, finora purtroppo rimasto solo teorico. "Mettere le mani in pasta" ci ha sicuramente rivelato degli aspetti quanto più possibile vicini all'esperienza reale che con ogni probabilità ci aspetta negli anni futuri.

Per quanto riguarda il suggerimento in coda alla consegna sull'utilizzo del framework di simulazione iCasa, le nostre scelte di modellazione, giorno per giorno, si sono scostate da un sereno utilizzo dello stesso, spingendoci verso la creazione di un simulatore ad hoc sul quale fare appoggio per il nostro software di gestione domotica.

Infine volevamo sottolineare le nostre impressioni sull'uso di GitHub come piattaforma di lavoro: difatti non ci siamo limitati al suo utilizzo come repository remota, ma abbiamo anche organizzato le sessioni di lavoro sulla base di task assegnati alla voce "issues". Per ognuno di essi abbiamo anche creato delle branch apposite per meglio gestire le modifiche dovute al progresso dell'implementazione. Ha sicuramente contribuito all'organizzazione e sincronizzazione del nostro lavoro, impedendo che si verificassero situazioni di perdita di tempo dovute a versionamenti incorretti e/o sovrapposizioni, che ci hanno aiutato a restringere le tempistiche in modo tale da soddisfare i termini di consegna senza affanni.

Il team di sviluppo di SmartHome,

Acronimi

CLI Command Line Interface. 20, 21

CP Conflict Pool. 20–22

EMAC Entity Monitor Automation Control. 2, 4, 21, 22

MQTT Message Queue Telemetry Transport. 21

SEC Secutity and Error Coverage. 2, 6, 20, 22

Bibliografia

[1] Electronjs. 19

[2] Alexander Shvets. *Dive Into Refactoring*. Refactoring.Guru, 2019. 31