Università degli studi di Bologna

Facoltà di Ingegneria e Scienze Informatiche

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Sistemi Distribuiti

# Smarthome Framework

**Studenti**

*Luca Maestri* 0900057223

*Gioele Masini* 0900056980

*Andrea Pruccoli* 0900057222

# Indice

# Introduzione

Negli ultimi anni la domotica è stata al centro di numerosi dibattiti e ha acquisito un numero sempre maggiore di attenzioni, come disciplina essa si occupa dello studio delle tecnologie volte a migliorare la qualità della vita nella casa e più in generale negli edifici. Le soluzioni tecnologiche che possono essere adottate per la realizzazione di un sistema domotico sono caratterizzate da peculiarità d'uso proprie degli oggetti casalinghi: semplicità, continuità di funzionamento, affidabilità, basso costo. Le tecnologie per la domotica permettono inoltre di ottenere alcuni vantaggi quali ad esempio: risparmio energetico, automatizzazione di azioni quotidiane. Il mercato ad oggi offre diverse soluzioni competitive ma i numerosi ostacoli lasciano ancora ampio spazio per innovare e proporre alternative alle tecnologie esistenti. A questo scopo è stato realizzato il seguente framework che intende muovere passi in avanti verso la definizione di un più semplice ambiente di sviluppo di dispositivi interconnessi in maniera sempre più agevole ed efficace.

# Requisiti

Si necessita di una serie di strumenti per avviare la produzione di dispositivi intelligenti di vario genere da impiegare in ambito di domotica. Ogni dispositivo deve poter essere installabile singolarmente ma deve anche essere in grado di coordinarsi e collaborare con altri dispositivi sviluppati con le stesse tecnologie per la fruizione di funzionalità più complesse.

Deve essere presente un’interfaccia utente per monitorare e configurare i dispositivi attivi e che faciliti lo sviluppo e i test di nuovi dispositivi.

# Analisi

L’analisi dei requisiti ha permesso di individuare i seguenti requisiti tecnici da soddisfare:

- **Autonomia**: ogni dispositivo dovrà funzionare a prescindere dai componenti disponibili in rete e dalla presenza della rete stessa.
- **Interazione e cooperazione**: i dispositivi in rete collaboreranno per fornire all’utente funzionalità avanzate, maggior precisione e/o risparmio di risorse. Sarà inoltre possibile interagire con il sistema tramite appositi client.
- **Scalabilità, openness e extensibility**: sarà semplice aggiungere nuovi e diversi dispositivi al sistema senza comprometterne il funzionamento. In questo modo sarà adatto a tutti gli ambiti e contesti, dal privato al professionale.
- **Fault tolerance**: resistenza ad errori e malfunzionamenti temporanei o permanenti. Il sistema sarà ottimizzato per gestire le problematiche più comuni dei sistemi distribuiti nel contesto scelto quali: il guasto di un dispositivo, la mancanza di connettività, il

superamento del limite di Kwh a disposizione, condizioni ambientali estreme (forte pioggia, vento, caldo/freddo, etc.).

# Progettazione

Si è scelto di realizzare un framework Java che permetta di gestire in maniera distribuita i dispositivi. Le tecnologie scelte sono Jade e Tucson in quanto le funzionalità fornite permettono di soddisfare i requisiti identificati in fase di analisi.

L'interfaccia verrà sviluppata anch'essa in Java tramite la libreria Swing per poter essere implementata in maniera indipendente dalla piattaforma di esecuzione, in modo che funzioni su ogni sistema operativo. Anch'essa farà uso di Tucson e Jade per una migliore integrazione con il sistema.

Di seguito si elencano tutte le entità modellate e le strategie definite in fase di progettazione.

## Entità

Le principali entità da cui è composto il framework sono le seguenti:

- Edificio
- Stanza
- Device

### Edificio

L'intero framework deve focalizzarsi sulla gestione di un singolo edificio. In fase di progettazione non è stato modellato concretamente in quanto viene rappresentato dal sistema stesso. Ogni edificio è composto da una o più stanze.

### Stanza

Ogni stanza rappresenta una porzione di edificio. La modellazione di questa entità permette di rappresentare in maniera virtuale lo spazio fisico e di raggruppare secondo criteri logici i dispositivi. È un elemento di coordinazione fondamentale ma non possiede logica in quanto sono i singoli dispositivi a fare uso di questa informazione quando necessario. Viene rappresentato da una tupla apposita sul tuple centre che contiene le seguenti informazioni: numero identificativo univoco generato automaticamente, nome della stanza e numero del piano dell'edificio in cui si trova la stanza. Una prima stanza in cui inserire di default i dispositivi viene caricata all'avvio del sistema.

### Device

L'elemento principale e il più delicato da gestire in fase di progettazione. Ogni dispositivo è rappresentato sul tuple centre da una tupla che ne contiene tutti i dati principali: un identificatore univoco, un nome personalizzabile, la classe del device, l'identificatore jade

dell'agente DeviceAgent (vedi paragrafo seguente) e l'id della stanza in cui si trova. Ogni dispositivo può essere assegnato ad una singola stanza e al primo avvio i dispositivi vengono automaticamente assegnati alla stanza di default.

Ogni dispositivo è composto dall'agente generico DeviceAgent, che implementa una serie di funzionalità basilari comuni a tutti i device, e da uno o più agenti che ne modellano i comportamenti specifici. Questa scelta permette al programmatore di sviluppare il proprio device occupandosi solo di implementare le sue funzioni specifiche e permette il riuso del codice. Rende inoltre immediata l'implementazione di dispositivi con lo stesso comportamento ma componentistica hardware differente o con un superset di funzionalità rispetto a dispositivi implementati precedentemente (si pensi ai singoli sensori i cui agenti possono essere riutilizzati nello sviluppo di una stazione meteo).

Il framework fornisce una o più classi astratte che implementano le funzionalità comuni a tutti i dispositivi.

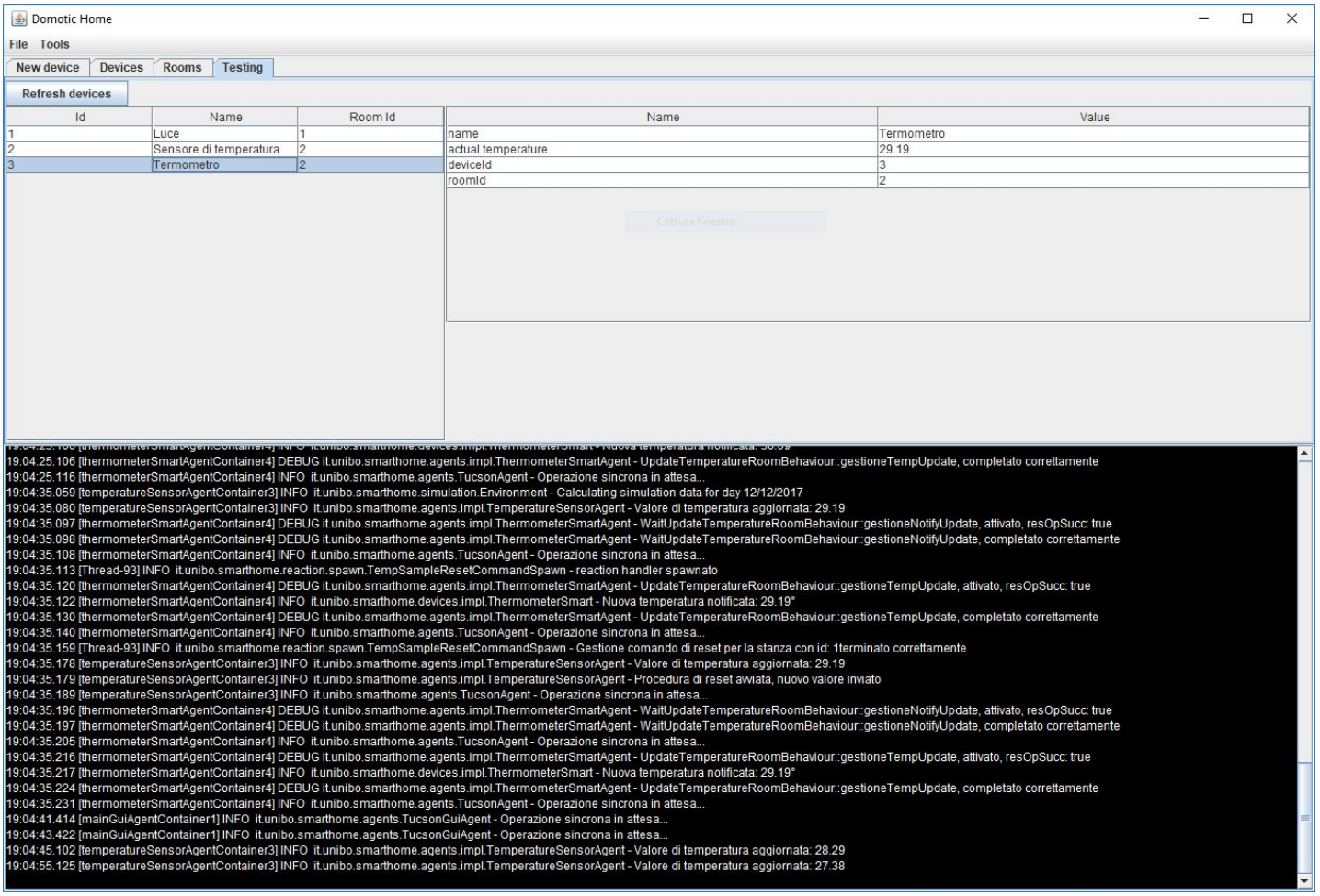
# Interfaccia utente

Le tecnologie e le scelte implementative hanno permesso la realizzazione di una interfaccia utente completamente autonoma e perfettamente integrata in un sistema così complesso.

Le funzionalità di cui dispone sono le seguenti:

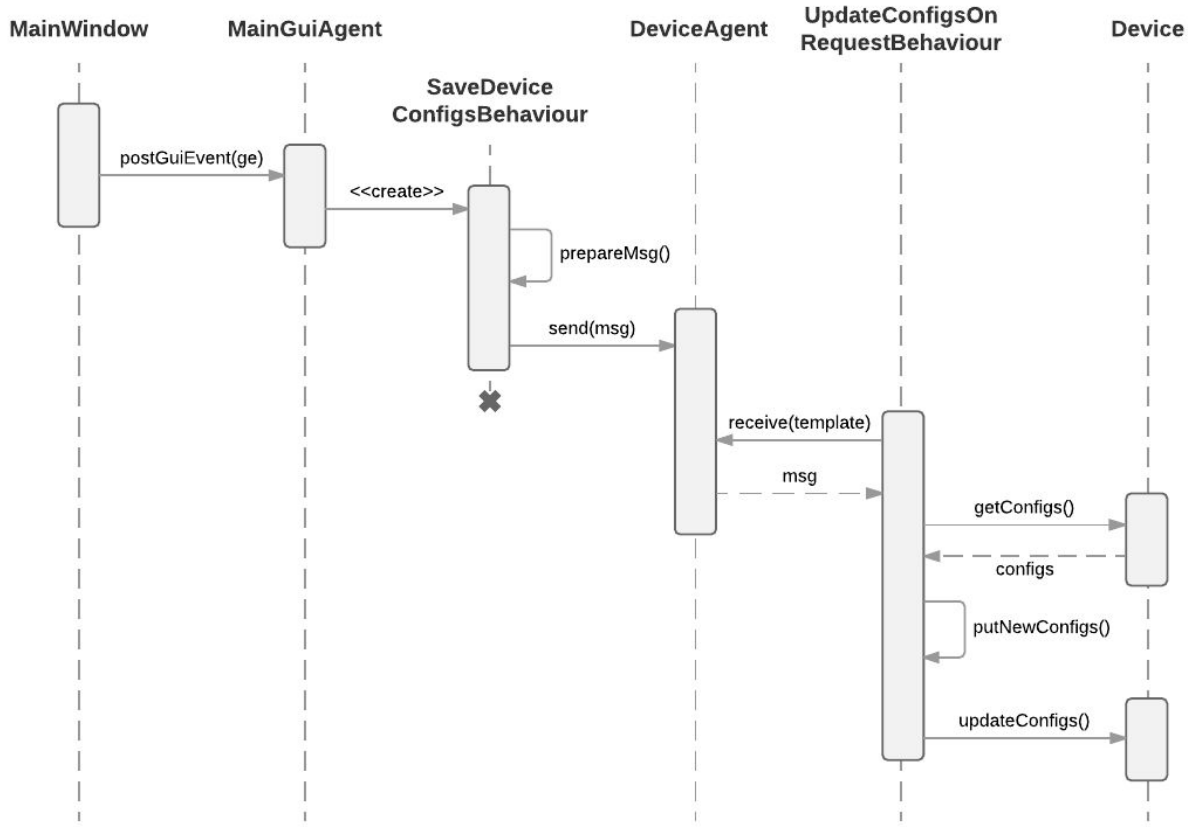
- **New device**: Istanziazione di nuovi dispositivi;
- **Devices**, **Testing**: Visualizzazione, configurazione, lettura dei dati e simulazione delle interazioni dell'utente dei dispositivi presenti in rete;
- **Rooms**: Gestione delle stanze;
- **Tools** menu: Collegamenti per l'avvio di tool per la gestione delle tecnologie alla base del framework: Tucson Inspector e Jade RMA.

Di seguito un'immagine che mostra l'interfaccia implementata.




Per integrare al meglio l'interfaccia con il sistema progettato si è scelto di implementarne il controller come agente estendendo la classe GuiAgent di Jade. Per ogni funzionalità è stata creata un'apposita classe estendente Behaviour in modo che sia riutilizzabile nell'eventuale implementazione di interfacce alternative. La comunicazione avviene sia tramite messaggi Jade che con l'uso del tuple centre, a seconda delle necessità. Ogni device avvia un proprio DeviceAgent, descritto nel dettaglio nel prossimo capitolo, che si occupa di mantenere attivi i behaviour necessari per gestire le richieste dell'interfaccia utente.

Si illustra con un diagramma di sequenza come avviene l'aggiornamento delle configurazioni di un device da interfaccia facendo uso dei messaggi Jade. È un flusso al quale possono essere ricondotte anche la maggior parte delle altre funzionalità.




# La coordinazione

La coordinazione tra i device viene gestita dagli agenti Jade che ne governano il funzionamento usando i mezzi messi a disposizione da Jade e Tucson: messaggi ACL e tuple centre.

Nello specifico i messaggi Jade sono stati scelti per i casi in cui è necessaria una comunicazione diretta verso un unico specifico agente, per le azioni di testing e simulazione della realtà in quanto sono assolutamente trasparenti e facilmente disabilitabili in un futuro ambiente di produzione.

Il tuple centre invece è stato utilizzato quando la coordinazione richiede che un dispositivo condivida parte dei suoi dati o il suo stato con il sistema. Questo approccio permette al singolo dispositivo di contribuire alla coordinazione degli elementi dell'edificio in modo costante e imprescindibile, senza la necessità di dover conoscere la composizione della rete. Per semplificare l'implementazione di queste azioni si è fatto uso frequente delle reactions che hanno permesso di rendere trasparenti alcune logiche implementative e semplificato la programmazione in java.

Per automatizzare la conversione di oggetti java in tuple e viceversa, e quindi per agevolare al programmatore l'uso di Tucson, sono state implementate delle classi DAO, una per ogni dato a cui corrisponde una tupla sul tuple centre, che contengono i metodi per generare le

tuple o per popolare l'istanza della classe a partire da una LogicTuple letta dal tuple centre. che le rappresentano interamente o parzialmente (tramite l'uso delle variabili Prolog) ed è anche possibile popolare i dati di un DAO a partire da una LogicTuple.

## La distribuzione

L'applicazione è stata progettata per essere completamente distribuibile con il solo requisito che gli agenti di un singolo device vengano istanziati sulla stessa macchina. Tutti gli agenti di un dispositivo vengono inoltre registrati all'interno di uno stesso contenitore, creato ad-hoc, ottenendo così una struttura ben organizzata. Grazie a tale scelta è stato inoltre possibile generare nomi univoci per gli agenti dei dispositivi con una strategia molto semplice: si concatena il nome generico dell'agente con il nome del contenitore.

Anche l'interfaccia utente è stata sviluppata con il paradigma ad agenti che le permette di essere istanziata su una qualsiasi macchina appartenente alla rete senza perdita di funzionalità.

# Funzionalità del framework

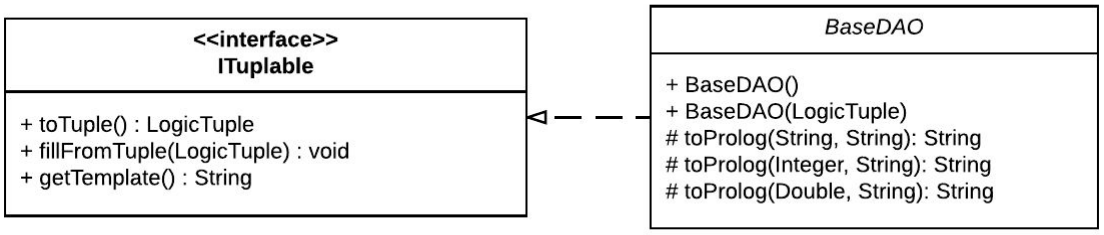
## Generazione di id univoci

Tramite Tucson e Respect si fornisce una funzionalità per generare id univoci progressivi, utilizzabile tramite il behaviour RdUniqueIdBehaviour. Ogni volta che viene richiesto un id si specifica un codice testuale che identifica la serie a cui appartiene il valore, in questo modo si possono generare diverse serie di identificatori, ognuna con la propria progressione. Ciò avviene tenendo registrato sul tuple centre il prossimo id disponibile per ogni serie e provvedendo a incrementarlo, con una reaction, dopo ogni lettura per renderlo pronto alla richiesta successiva. Le serie vengono inizializzate tramite una seconda reaction che si attiva sull'invocazione di ogni lettura: essa verifica l'assenza della tupla della serie richiesta e in tal caso la crea prima della lettura con valore 1, inizializzando in questo modo la progressione.

## ITuplable

Interfaccia che ogni DAO deve implementare.

Per guidare e semplificare la creazione di un DAO è stata implementata la classe astratta BaseDAO che offre delle funzioni per gestire i singoli attributi dell'oggetto, ed eventualmente sostituirne il valore con una variabile Prolog. BaseDAO implementa due costruttori: il primo senza parametri, per le interrogazioni Tucson, il secondo per popolare l'oggetto con i dati di una tupla compatibile.

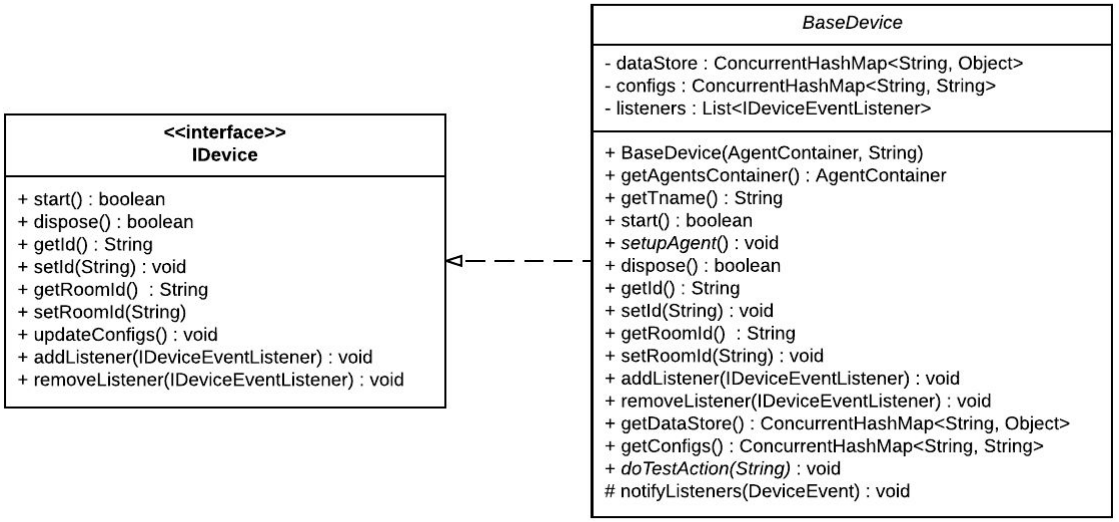



I DAO sono molto utili per la gestione dei dati su tuple centre in quanto, oltre a contenere i valori che compongono il dato, possiedono le seguenti funzionalità:

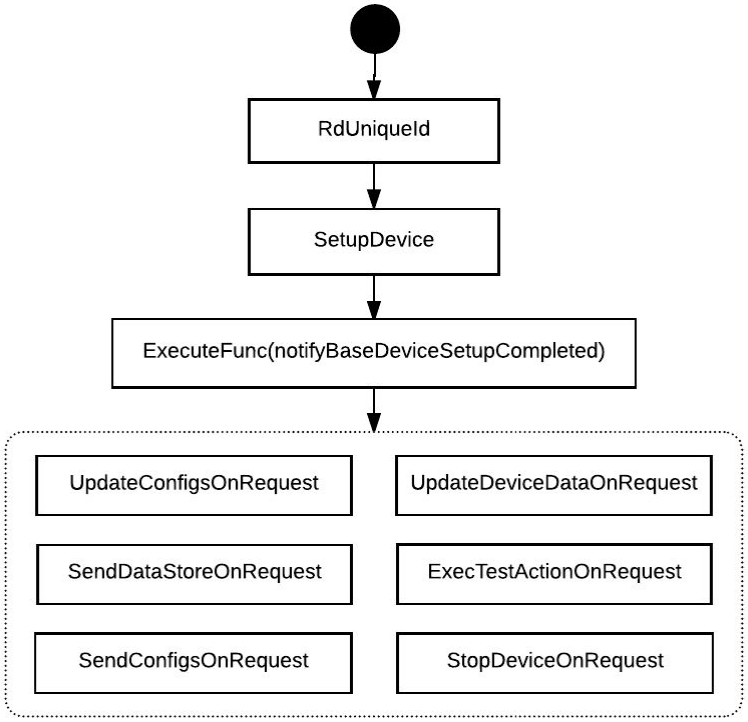
- Generazione della tupla contenente i singoli dati, con la possibilità di sostituirne uno o più con variabili Prolog (metodi `toTuple` e `asTupleString`);
- Popolazione delle variabili di un DAO con i valori di una tupla corrispondente (metodo `fillFromTuple`).

## BaseDevice

La classe più importante fornita dal framework. Contiene il datastore, una mappa thread-safe di oggetti usata per condividere dati tra gli agenti, e configs, un'altra mappa thread-safe in cui vengono registrate tutte le impostazioni del device. Questi due oggetti comuni a tutti i device hanno permesso una semplice implementazione delle tab Device e Testing dell'interfaccia utente. Questa classe permette anche agli agenti di registrarsi come listener sull'esecuzione di particolari eventi definibili per ogni device.




Infine fornisce le funzionalità base che caratterizzano l'intero sistema, attraverso l'agente DeviceAgent. Questi avvia due behaviours: il primo si occupa di eseguire in sequenza tutte le azioni necessarie al setup del dispositivo collegato, il secondo si occupa di regolare, in modo parallelo, i behaviour per gestire e/o rispondere ai messaggi della gui.




La prima azione che il DeviceAgent compie è ottenere un identificatore univoco dal tuple centre e assegnarlo al device che gestisce, fatto ciò è in grado di avviare SetupDeviceBehaviour per generare la tupla *device* e scriverla sul tuple centre. L'ultima azione di setup è eseguita dal behaviour ExecuteFunc, esso notifica al device di aver completato la procedura di configurazione iniziale. Una volta terminata questa fase iniziale, l'agente manterrà attivi in modo parallelo una serie di behaviour che gli permettono di rimanere in ascolto di specifici messaggi, alla ricezione dei quali esegue una delle seguenti azioni:

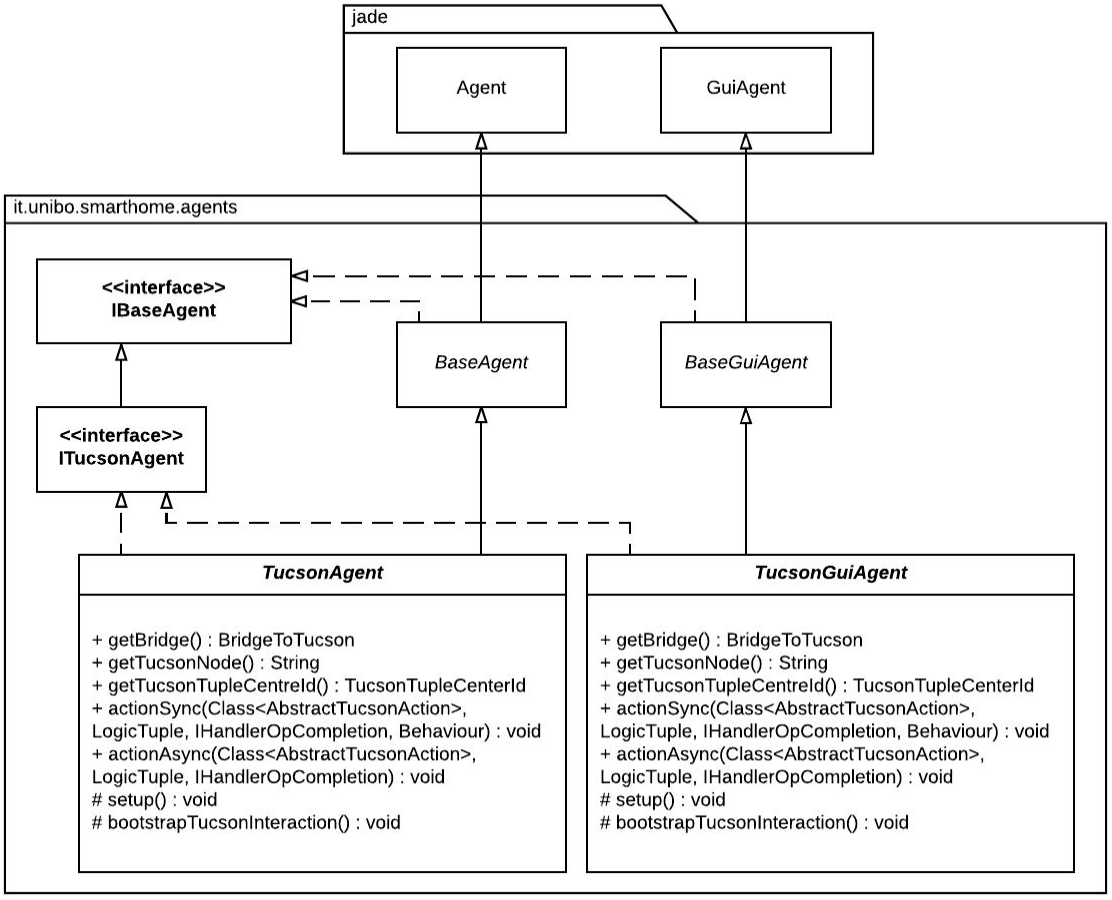
- Aggiornamento della configurazione;
- Aggiornamento dei dati base (nome e id stanza) del device;
- Invio di dati presenti sul datastore;
- Esecuzione di un'azione di testing;
- Invio della configurazione corrente;
- Stop del device.

I behaviour elencati rimarranno tutti attivi fino allo spegnimento del device e permettono la maggior parte delle interazioni dell'utente con la gui.

## TucsonAgent e TucsonGuiAgent

Implementazioni dell'interfaccia ITucsonAgent, permettono di istanziare un agente Jade dotato di funzionalità per l'invocazione di azioni Tucson sincrone o asincrone sul tuple centre. Vengono forniti due metodi di nome actionSync e actionAsync in diverse varianti,

differenziate per il tipo di dato usato per rappresentare le tuple in input. Si supportano tuple di tipo String e LogicTuple e classi DAO. La logica di conversione è gestita internamente.




Ognuno di questi agenti inoltre viene inizializzato con i dati necessari per la comunicazione verso il tuple centre: il tuple centre id, il TucsonHelper con ACC e il BridgeToTucson.

A loro volta entrambe le classi estendono BaseAgent e BaseGuiAgent, pensati per fornire sole funzionalità per il framework Jade.

Per evitare duplicazioni di codice è stata scritta una classe TucsonOpUtils in cui sono state implementate concretamente tutte queste funzionalità.

# Implementazioni

A fini dimostrativi e di testing sono stati implementati una serie di device che fanno uso del framework sviluppato e fungono da esempio per il suo utilizzo. Tutti i dispositivi implementati simulano l'interazione con il relativo hardware.

## Luce

Classe LightSimple.

Implementazione di una semplice luce, può essere accesa o spenta. Tramite configurazioni è possibile cambiare la modalità della luce da manuale (default) ad automatica.

L'agente a cui è demandato il compito di accendere e spegnere la luce in modo automatico è il LightAgent. Al suo interno implementa un FSMBehaviour a due stati, ed ogni stato è un SimpleBehaviour: il primo, AutomaticTurnLightOnBehaviour, rimane in attesa di leggere dal tuple centre uno o più tuple di movimento riguardanti la stanza in cui è installata la luce, quando questo avviene imposta lo stato della luce ad accesa; il secondo behaviour della finite state machine, AutomaticTurnLightOffBehaviour, rimane in attesa della cancellazione dal tuple centre delle tuple di movimento relativa alla stanza, quando ciò avviene imposta lo stato della luce a spenta.

## Sensore di temperatura

Classe SimulatedTemperatureSensor.

Permette di leggere la temperatura della stanza. Per non restituire valori completamente casuali e disomogenei, a fine di testing è stata creata una classe Environment che genera valori consistenti tra di loro in un range definito a priori.

L'agente che ne governa il funzionamento, TemperatureSensorAgent, è composto da due behaviour paralleli.

Il primo è un TickerBehaviour che si occupa di leggere la temperatura rilevata dal sensore e, se vi sono variazioni significative, registrare la relativa tupla sul tuple centre. Ogni volta che viene quindi registrato il valore della temperatura rilevato da un sensore viene automaticamente aggiornata una seconda tupla  contiene la temperatura media della stanza, calcolata sulla base di tutti i sensori registrati all'interno della stessa. Per evitare che guasti improvvisi a un sensore impediscano la rimozione dei dati relativi ad esso, lasciando quindi informazioni non aggiornate che rimangano all'interno del tuple centre, alterando il calcolo del valore medio, è stato implementato un meccanismo grazie a Respect, che permette di svuotare, in base alla frequenza con il quale viene letto il valore di temperatura media e al numero di sensori presenti, i dati registrati fino a quel momento.

In questo contesto entra quindi in azione il secondo behaviour, un CyclicBehaviour che, in caso di reset esterno dei dati del sensore, si occupa di ricaricare i dati aggiornati all'interno del tuple centre, in questo modo si mantiene consistente e aggiornato il valore della temperatura media della stanza, nonostante eventuali guasti non previsti.

## Termometro

Classi ThermometerSimple, ThermometerSmart.

Per svolgere la funzionalità di visualizzazione della temperatura media di una stanza, calcolata grazie alle misurazioni dei sensori di temperatura, sono stati implementati due diversi device: ThermometerSimple e ThermometerSmart. Questo è stato fatto per mostrare due possibili modalità differenti di utilizzo del sistema che esso offre.

ThermometerSimple: si serve del ThermometerSimpleAgent, un agente provvisto di TickerBehaviour che periodicamente effettua una lettura della temperatura media della stanza dal tuple centre, una volta recuperato il valore si occupa di notificare il device che dovrà limitarsi ad aggiornare la temperatura letta.

ThermometerSmart: basa il suo funzionamento sull'agente ThermometerSmartAgent, il quale utilizza un FSMBehaviour composto da due stati che vengono ripetuti in maniera ciclica. Nel primo stato il WaitUpdateTemperatureRoomBehaviour attende la notifica di un aggiornamento della temperatura media dal tuple centre. Questa tupla viene scritta per ogni device che è interessato a ricevere aggiornamenti sulla variazione della temperatura di una precisa stanza, essa viene aggiunta al tuple centre dalla stessa reaction che, come precedentemente descritto nel paragrafo riguardante il sensore di temperatura, si occupa di calcolare la temperatura media della stanza in base ai valori dei singoli sensori presenti all'interno di essa. Una volta ricevuta la notifica si passa allo stato successivo dove il behaviour UpdateTemperatureRoomBehaviour si occupa di effettuare la lettura del valore della temperatura media aggiornato, lo comunica al device ed infine effettua una nuova richiesta di notifica di aggiornamento della temperatura della stanza.

## Telefono

Classe PhoneSimple.

Il device PhoneSimple rappresenta una semplice implementazione di un dispositivo adibito alle classiche funzioni di comunicazione verso l'esterno tramite linea telefonica. Esso permette l'invio di messaggi a qualsiasi entità interessata a questa funzione. Grazie al behaviour MessageBehaviour, dell'agente PhoneAgent, recupera dal tuple centre le richieste di invio di messaggi, i relativi dati necessari e si occupa di comunicarli al device il quale provvede a soddisfare la richiesta.

## Sensore di movimento interno

Classe SimulatedMotionSensor.

Il device SimulatedMotionSensor rappresenta l'implementazione di un sensore di movimento che registra se c'è o meno del movimento all'interno della stanza in cui si trova. Quando viene rilevato del movimento si informa il sistema tramite la scrittura sul tuple centre di una tupla di tipo MovementRoomDAO da parte del WriteMovementBehaviour (estende OneShotBehaviour). I dispositivi interessati a questa informazione potranno leggerla usando Tucson. Quando non viene più registrato del movimento nella stanza, la tupla

precedentemente inserita viene cancellata dal CleanMovementBehaviour (estende SimpleBehaviour).

Questi behaviours vengono istanziati e avviati al bisogno dall'agente MotionSensorAgent in risposta all'evento che notifica la presenza/assenza di movimento inviato dal dispositivo.

## Finestra allarmata

Classe WindowAlarmedSmart.

Il device in questione rappresenta una finestra dotata di un sensore che rileva l'apertura e la chiusura della stessa. Grazie all'interazione con il sistema, può estendere le proprie funzionalità di base interagendo con un dispositivo di *Allarme*. Infatti nel caso in cui sia attivo il sistema di allarme della casa, il sistema verrà notificato dell'apertura della finestra, in questo modo l'allarme potrà svolgere le sue mansioni supportato da questo dispositivo.

A questo dispositivo sono collegati due agenti:

- WindowAlarmStateAgent: è responsabile di allarmare o disallarmare la finestra coerentemente allo stato in cui si trova l'edificio (governato dal dispositivo di *Allarme*, come indicato successivamente).
- SendAlarmAgent: estende un CyclicBehaviour, il suo compito è quello di notificare il sistema nell'eventualità che la finestra venga aperta mentre il dispositivo è allarmato.

## Allarme

Classe AlarmSimple.

Implementazione dell'allarme dell'edificio. Possiede diversi parametri modificabili nelle configurazioni: il tempo per cui il segnale di allarme viene mantenuto attivo, il messaggio e il numero di emergenza da utilizzare quando viene avviato il protocollo di allarme.

Il dispositivo fa uso di AlarmAgent, agente a cui è demandato il compito di impostare lo stato dell'edificio come allarmato/non allarmato e di eseguire le operazioni di sicurezza (attivare i segnalatori acustici ed avvertire le autorità competenti, ad esempio) in risposta agli eventi di effrazione.

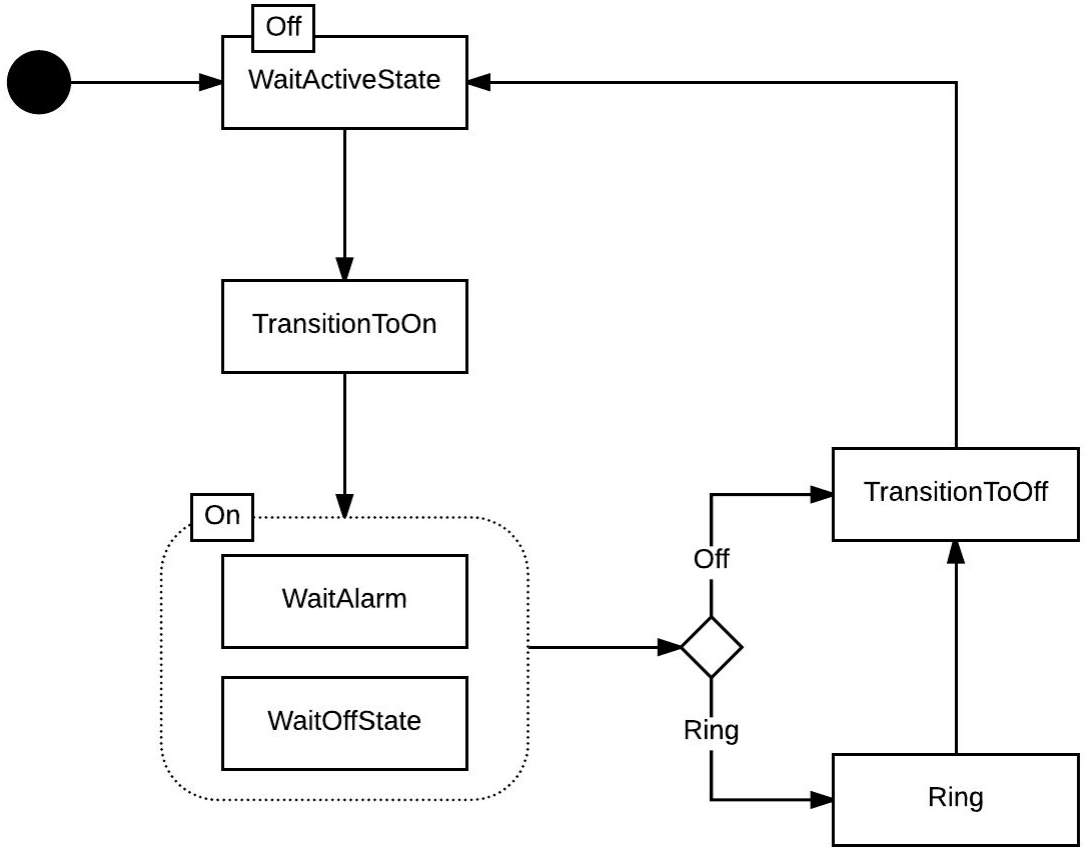
La notifica di un'effrazione può essere notificata direttamente dai dispositivi allarmabili (come la finestra, nel caso di questo progetto), oppure indirettamente tramite l'uso di reaction da parte di questi dispositivi che non fanno del controllo della sicurezza il loro obiettivo primario (è questo il caso del sensore di movimento, il quale compito non è di certo quello di comunicare direttamente con l'allarme in caso di movimento nella stanza in cui esso è installato).

L'agente implementa un FSMBehaviour che effettua i seguenti step:

1. Rimane in attesa di leggere, dal tuple centre, la tupla che avvisa il device di impostare lo stato di edificio allarmato (WaitActiveStateBehaviour);

2. Pulisce il tuple centre da eventuali tuple che segnalano un'effrazione e scrive la tupla che indica che l'edificio è stato allarmato (TransitionToOnBehaviour);
3. Ora l'allarme rimane in attesa di due eventi (OnBehaviour, un ParallelBehaviour):
    a. Riceve la notifica che vi è stata un'effrazione del perimetro (WaitAlarmBehaviour), avvia dunque il protocollo di allarme che prevede l'invio di un messaggio al numero di emergenza (se c'è un dispositivo PhoneSimple) e l'avviso dell'effrazione ai presenti nella casa (ma anche a chi è all'esterno) tramite un segnale acustico;
    b. Riceve l'ordine di disallarmare l'edificio (WaitOffStateBehaviour).
4. Se è stato avviato il protocollo di allarme, l'FSMBehaviour, dall'OnBehaviour passa al RingBehaviour (estende WakerBehaviour), il quale ha il compito impostare lo stato dell'edificio a non allarmato in modo automatico dopo che è trascorso un determinato periodo di tempo (quello impostabile nelle configurazioni);
5. Sia che sia stato avviato il protocollo d'allarme o meno, prima di ritornare allo step 1, l'FSMBehaviour passa per il TransitionToOffBehaviour, il quale pulisce il tuple centre da eventuali tuple che segnalano un'effrazione e dalla tupla che indica che l'edificio è stato allarmato; dopo queste azioni, l'FSMBehaviour ritorna allo stato di WaitActiveStateBehaviour.

Di seguito una raffigurazione grafica dei Behaviours tramite diagramma a stati.



# Avvio e uso del software

E' possibile avviare il programma con i seguenti parametri:

- **`--main`**, è la prima opzione da usare, istanzia il MainContainer di Jade, il tuple centre e il SettingsAgent che carica le reactions e inserisce la tupla di una generica stanza in cui posizionare i device quando creati;
- **`--gui`**, istanzia l'interfaccia utente. E' possibile istanziarla su una macchina diversa da quella del main indicandone l'indirizzo IP tramite l'opzione --tc;
- **`--device <device class full name>`**, richiede l'istanziazione del device indicato come argomento. Ogni device è istanziato in un container jade apposito, l'uso di questo parametro richiede quindi che in precedenza o in concomitanza sia stato creato il tuple centre e il MainContainer tramite l'opzione --main;
- **`--tc <IP address>`**, permette di specificare l'indirizzo in cui è stato istanziato il main del programma. E' un attributo obbligatorio per poter avviare l'applicazione su una macchina diversa in maniera distribuita;

Avviare l'applicazione senza parametri corrisponde a usare i flag `--main --gui`. I parametri vengono gestiti dalla classe ProgramArgumentsHelper che nasconde la logica di parsing.

Di seguito vengono indicati alcuni esempi.

*Primo avvio del programma*

```
smarthome.jar --main
```

*Istanziazione del device LightSimple sulla stessa macchina del main*

```
smarthome.jar --device it.unibo.smarthome.devices.impl.LightSimple
```

*Istanziazione del device LightSimple sulla macchina locale mentre il main è attivo all'indirizzo 192.168.1.13 e apertura della gui*

```
smarthome.jar --gui --device
it.unibo.smarthome.devices.impl.LightSimple --tc 192.168.1.13
```

# Difficoltà incontrate

Durante lo sviluppo è stato necessario dedicare del tempo a studiare la risoluzione di un problema di concorrenza rilevato nella libreria Tucson4Jade. Si verificava frequentemente un errore bloccante, dovuto a una corsa critica, quando venivano effettuate operazioni sincrone sul tuple centre. La questione è stata risolta sincronizzando la classe SynchCompletionBehaviourHandler di Tucson4Jade.

# Possibili miglioramenti

Grazie alla forte ingegnerizzazione del software e all'approccio utilizzato durante lo sviluppo del progetto, il sistema, già completo, può essere esteso in numerosi modi, tra i possibili miglioramenti elenchiamo alcune possibili modifiche che permetterebbero l'introduzione di nuove interessanti funzionalità e una serie di applicazioni concrete valide per ogni contesto.

## Nuove funzionalità del framework

Con modifiche al framework sarà possibile aggiungere funzionalità all'intero sistema investendo del tempo laddove sono state adottate semplificazioni per giungere all'ottenimento di una prima versione stabile in tempi consoni alla presentazione del progetto d'esame.

Innanzitutto la comunicazione tra i diversi agenti di un singolo device può essere implementata con l'uso di jade e tucson come tecnologie di comunicazione e coordinamento invece dell'attuale sistema più object-oriented. Ciò eliminerebbe il vincolo posto in fase di progettazione permettendo agli agenti di spostarsi fra le macchine della rete per, ad esempio, bilanciare carichi computazionali particolarmente pesanti.

Una seconda possibilità è quella di distribuire il tuple centre per favorire così la scalabilità del sistema e l'uso del framework in ambiti professionali più complessi di una semplice abitazione. Con lo stesso scopo si potrebbero inoltre modellare gli edifici e affiancare alle stanze (luoghi interni all'edificio) una rappresentazione virtuale dell'esterno per meglio gestire giardini e spazi aperti.

Relativamente al testing si potrebbe inoltre sviluppare in modo distribuito la classe Environment in modo da poter programmare test più o meno automatici e, soprattutto, distribuiti. Sempre con finalità di test e debug si potrebbe, tramite un'opzione, creare una configurazione per avviare il framework in modalità simulazione, in cui i dati o anche interi dispositivi vengono simulati per poter eseguire test in fase di sviluppo ma anche allo scopo di individuare e risolvere problemi che si verificano solo in determinati ambienti reali.

## Dispositivi e agenti implementabili

Un agente implementabile e che potrebbe essere fornito insieme al framework stesso per la sua elevata riusabilità è il timer. Collegato ad un dispositivo orologio, all'orologio virtuale di una macchina o a quello di un servizio web può fornire una serie di funzionalità molto utili a qualsiasi sistema per temporizzare l'esecuzione di specifiche azioni: pianificare e/o misurare i tempi di avvio, spegnimento e durata, solo per fare alcuni esempi.

Un altro agente importante è quello che monitora e condivide le condizioni meteo. Oltre ai dati sulla temperatura già gestiti nella nostra implementazione si dovrebbero misurare velocità e direzione del vento, precipitazioni atmosferiche, umidità e quantità di luce sia

all’interno che all’esterno degli edifici. Tali informazioni, almeno per l’esterno, potrebbero anche essere ottenute dai servizi web meteo per ridurre i costi in hardware e manutenzione.

Fra le ulteriori, innumerevoli, possibilità sicuramente spiccano quelle relative alla sicurezza (di cui è stato implementato un semplice esempio) e alla sostenibilità ambientale dell’abitazione.

Aggiungere la possibilità di passare parametri alla classe java nel metodo spawn(exec(it.unibo.MyJavaClass.class)) dal secondo parametro in poi. Questo ci permetterebbe di fare la media della temperatura divisa per stanza.

La modifica andrebbe fatta aggiungendo dei getter e setter per gli argomenti in alice.tucson.api.AbstractSpawnActivity e il metodo spawn(exec()) andrebbe esteso modificando la funzione spawnActivity presente nella classe alice.respect.core.RespectVMContext. Gli argomenti potrebbero essere gestiti semplicemente come TupleArgument e sarà poi onere della classe java che estende AbstractSpawnActivity ottenerne i valori.