Corso di laurea magistrale in
Ingegneria e Scienze Informatiche

Paradigmi di Programmazione e Sviluppo

iSerra

Framework per la gestione di serre intelligenti

Battistini Ylenia
Letizi Simone
Luffarelli Marta
Petreti Andrea

30 settembre 2020

Indice

1 Introduzione

Il progetto nasce con l'idea di simulare un committente proprietario di un'azienda agricola che vuole rinnovare il suo sistema di serre e semplificare il lavoro dei propri braccianti. Le esigenze del cliente, ipotizzate a seguito di un breve studio del dominio, guideranno la raccolta e la definizione dei requisiti.

In questo documento si descrive lo sviluppo di un framework per la gestione di serre intelligenti. Una serra intelligente è da intendersi come un ambiente in grado di autoregolarsi e di mantenere un micro-clima ottimale per la crescita delle piante. Le condizioni climatiche all'interno della serra (temperatura, umidità, luminosità, umidità del terreno,..) sono continuamente monitorate da sensori. Piccole variazioni in queste condizioni faranno scattare delle azioni automatiche in risposta tramite degli attuatori. Il nostro sistema consente la gestione di una serra territorialmente estesa e divisa in zone. Ogni zona può contenere diverse tipologie di sensori e attuatori. Il framework realizzato si compone di:

- una parte utile a creare un'istanza di una serra intelligente;
- una parte utile a interagire con la serra istanziata.

È strutturato, infatti, per rispondere alle esigenze di coloro che lo sfrutteranno per realizzare la loro implementazione di serra intelligente e per coloro che ricopriranno il ruolo di utenti finali della serra intelligente creata. In particolare, con riferimento al committente simulato, l'utente finale è individuato nel bracciante dipendente dell'azienda che avrà un accesso limitato alla gestione della serra; lo sviluppatore, invece, corrisponde ai dipendenti del settore informatico dell'azienda che si occuperanno della gestione più tecnica, determinando il comportamento della serra intelligente che creeranno.

2 Processo di sviluppo

Il processo di sviluppo adottato in questo progetto è di tipo incrementale ed iterativo, in particolare segue la metodologia Scrum. Il Product Owner del team, nonché sviluppatore, è Andrea Petreti, mentre il ruolo dello Scrum Master viene ricoperto a rotazione dagli altri membri. Ogni sprint ha la durata di una settimana (cinque giorni lavorativi) e se ne prevedono sei totali.

La prima parte di analisi e progettazione è stata svolta con la presenza fisica di tutti i membri per favorire la comunicazione, la successiva è stata svolta per via telematica.

2.1 Meeting

Ogni sprint vede il susseguirsi di diversi momenti, sia individuali che collettivi, in cui si discute di cosa si è fatto, di cosa si dovrà fare e chi lo dovrà fare e di eventuali problematiche riscontrate. In particolare sono previsti i seguenti meeting:

- *stand up meeting*, incontri giornalieri della durata massima di 15 minuti nei quali i membri comunicano i progressi effettuati nel giorno precedente, il piano di sviluppo per la giornata corrente ed eventuali problematiche emerse. Problematiche strutturali, o che comunque coinvolgono il lavoro di altri membri, andranno dettagliate separatamente in un altro meeting;
- *problem resolution meeting*, riunioni alle quali parteciperà solamente chi è coinvolto nel problema al fine di trovare possibili soluzioni che verranno riportate all'intero team;
- *sprint planning*, riunione della durata massima di un'ora, effettuata all'inizio di ogni nuovo sprint o alla fine, in concomitanza con la *retrospective*. In questa riunione si identifica cosa si vuole ottenere nello sprint successivo e si raffina il Product Backlog dettagliando i task da svolgere e assegnandoli ai membri del team;
- *retrospective*, incontro della durata massima di un'ora, volto a discutere lo sprint appeno concluso, a valutare ciò che ha funzionato e ciò che non è andato come programmato, cercando di trovare soluzioni ad hoc;
- *sprint review*, riunione della durata massima di un'ora effettuata in itinere durante uno sprint qualora venissero notati dei ritardi nella schedula e ci fosse l'esigenza di riassegnare task o rischedulare attività.

2.2 Revisione dei task

Il lavoro di implementazione procederà individualmente a meno di occasionali quality check in cui i membri non coinvolti nell'implementazione potranno valutare il lavoro svolto dai propri colleghi e proporre eventuali miglioramenti. Alcuni task di fondamentale importanza e interdipendenti verranno realizzati in modalità *Pair Programming*.

2.3 Tool

Il team ha concordato l'utilizzo di diversi strumenti a supporto dell'intero processo di progettazione e sviluppo al fine di efficientare il lavoro svolto. In particolare vengono utilizzati i seguenti tool:

- *Gradle* come strumento di build automation;
- *Travis CI* come strumento per la continuous integration;
- *AkkaTest e ScalaTest* come framework di testing;
- *GitHub* come repository Git;
- *GitHub Project* per la gestione del Product Backlog.

3 Requisiti

In questa sezione vengono descritti ed elencati i requisiti emersi in fase di analisi, elencati tenendo conto delle due diverse prospettive possibili: quella dell'utente finale (e.g. bracciante) che interagisce con il sistema e quella dello sviluppatore che estende il framework per realizzare una specifica serra. Segue un elenco dettagliato suddiviso in requisiti di *business*, *utente*, *funzionali*, *non funzionali* e *di implementazione*.

3.1 Business

I requisiti di business esprimono la "business solution" per un progetto, includendo i bisogni dei clienti e le loro aspettative[1]. A fronte di una prima fase di scoping sono emersi i seguenti requisiti di business:

1. creazione e gestione di una serra intelligente senza preoccuparsi di aspetti di basso livello
    - 1.1 integrazione di sensori e attuatori di varie tipologie
    - 1.2 suddivisione logica in zone al fine di consentire un controllo più peculiare sulle misurazioni di sensori e attuatori
    - 1.3 definizione semplice e intuitiva di regole che preservino le condizioni ottimali dell'intera serra
    - 1.4 monitoraggio e controllo automatizzati e continui tramite regole
    - 1.5 astrazione dalle logiche di comunicazione presenti tra sensori/ attuatori e serra
2. accesso remoto per interagire con la serra istanziata

3.2 Utente

I requisiti utente fanno riferimento ai bisogni degli utenti e descrivono quali sono le azioni che l'utente deve essere in grado di attuare sul sistema. La raccolta dei requisiti utente è riportata mediante *User Stories* e *Use Case Diagram*.

3.2.1 User Stories

La tabella 1 mostra i requisiti utente, sia lato bracciante che tecnico.

	UserStory-1
WHO	In qualità di tecnico o bracciante
WHAT	vorrei poter accedere in lettura allo stato delle zone
WHY	in modo che mi risulti semplice valutare azioni correttive

[1] https://www.isixsigma.com/implementation/project-selection-tracking/business-requirements-document-high-level-review/

	UserStory-2
WHO	In qualità di tecnico
WHAT	vorrei poter aggiungere delle regole specificando un insieme di parametri da valutare e le azioni conseguenti
WHY	al fine di mappare i processi decisionali tradizionali
	UserStory-3
WHO	In qualità di tecnico
WHAT	vorrei poter rimuovere le regole specificate
WHY	al fine di rimodulare il processo decisionale
	UserStory-4
WHO	In qualità di bracciante
WHAT	vorrei poter disabilitare una regola specificata
WHY	in modo da poter rispondere ad esigenze contingenti
	UserStory-5
WHO	In qualità di bracciante
WHAT	vorrei poter abilitare una regola specificata
WHY	in modo da poter ripristinare il processo decisionale
	UserStory-6
WHO	In qualità di tecnico
WHAT	vorrei poter aggiungere un sensore al sistema
WHY	in modo da poterlo associare ad una zona e percepirne i valori
	UserStory-7
WHO	In qualità di tecnico
WHAT	vorrei poter aggiungere un attuatore al sistema
WHY	in modo da poterlo associare ad una zona e fargli effettuare azioni
	UserStory-8
WHO	In qualità di tecnico o bracciante
WHAT	vorrei poter rimuovere un sensore dal sistema
WHY	in modo da poter sopperire ad eventuali guasti
	UserStory-9
WHO	In qualità di tecnico o bracciante
WHAT	vorrei poter rimuovere un attuatore dal sistema
WHY	in modo da poter sopperire ad eventuali guasti
	UserStory-10
WHO	In qualità di tecnico o bracciante
WHAT	vorrei poter aggiungere una zona
WHY	al fine di aumentare la capillarità delle percezioni e per attuazioni localizzate
	UserStory-11
WHO	In qualità di tecnico o bracciante
WHAT	vorrei poter rimuovere una zona
WHY	al fine di poter rimodulare zone esistenti
	UserStory-12
WHO	In qualità di tecnico o bracciante

WHAT	vorrei poter visualizzare le zone presenti nella serra
WHY	in modo da aver chiaro com'è organizzata al momento
	UserStory-13
WHO	In qualità di tecnico o bracciante
WHAT	vorrei poter associare un'entità ad una zona
WHY	in modo da ottenere maggiore capillarità in quella zona
	UserStory-14
WHO	In qualità di tecnico o bracciante
WHAT	vorrei poter dissociare un'entità da una zona
WHY	in modo da poter sopperire ad eventuali guasti o spostarla
	UserStory-15
WHO	In qualità di tecnico o bracciante
WHAT	vorrei poter accedere alle regole definite nella serra
WHY	in modo da poter decidere quali disabilitare/ abilitare

Tabella 1: User stories

3.2.2 Use Case Diagrams

In figura 1 è presente il diagramma UML dei casi d'uso relativo all'utente finale (il bracciante) che ha accesso limitato al sistema serra implementato. Il contesto indicato dal diagramma è la porzione di sistema esposta dal nostro framework con cui l'utente può interfacciarsi.

In figura 2 è presente il diagramma UML dei casi d'uso relativo allo sviluppatore utilizzatore del framework che implementa la sua serra intelligente. Il contesto indicato dal diagramma è l'intero sistema creabile sfruttando il nostro framework.

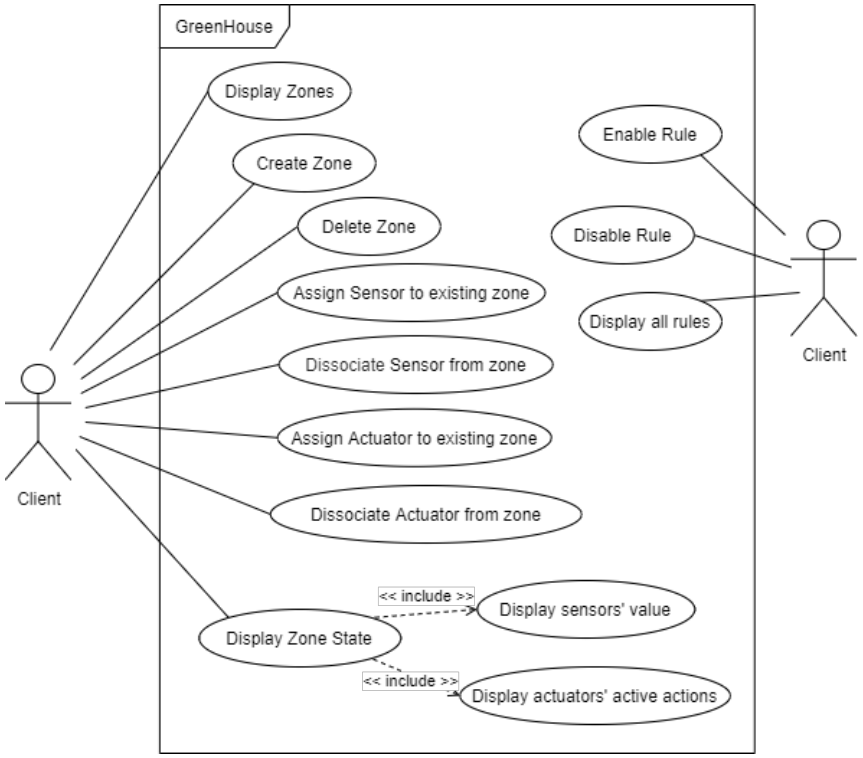

Figura 1: UML use case diagram - lato utente

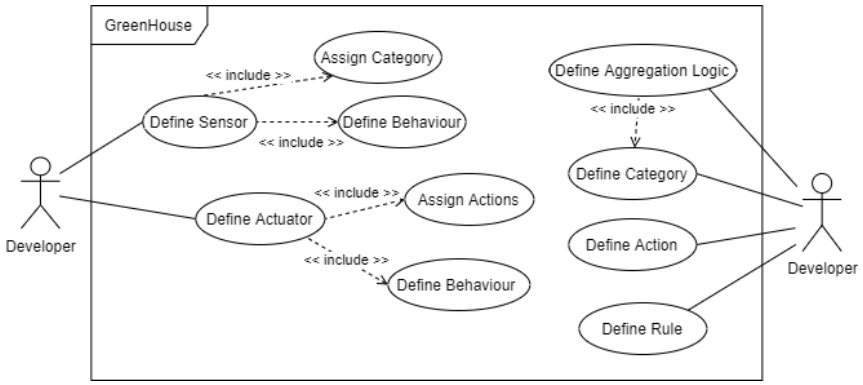

Figura 2: UML use case diagram - lato sviluppatore

3.3 Funzionali

I requisiti funzionali riguardano le funzionalità fornite dal nostro framework e sono estrapolati a partire dai requisiti utente individuati. In tabella 2 sono riportati quei termini che possono essere non chiari o fraintendibili. Segue l'elenco di tali requisiti.

1. Prendere decisioni per ripristinare lo stato ottimale della serra valutando regole esistenti descritte al suo interno
    - 1.1 eseguire le azioni solo sulle zone che hanno attuatori in grado di eseguirle
    - 1.2 eseguire le azioni definite nelle regole a fronte delle percezioni dei sensori
2. consentire l'aggiunta di regole
    - 2.1 definire le condizioni di attivazione che rappresentano le percezioni dei sensori
    - 2.2 definire le azioni che rappresentano ciò che possono fare gli attuatori
3. consentire la rimozione di una regola
    - 3.1 specificare la regola da rimuovere
4. consentire l'abilitazione di una regola disabilitata
5. consentire la disabilitazione di una regola abilitata
6. consentire l'aggiunta di sensori all'interno del sistema
    - 6.1 specificare una categoria di appartenenza del sensore
    - 6.2 consentire di definire il comportamento del sensore
7. consentire l'associazione di un sensore ad una e una sola zona esistente
    - 7.1 notificare il sensore dell'associazione avvenuta
8. consentire la dissociazione di un sensore dalla zona cui è assegnato
    - 8.1 ignorare le percezioni dei sensori non associati a zone
    - 8.2 notificare il sensore della dissociazione avvenuta
9. consentire la definizione di logiche di aggregazione dei valori dei sensori
10. consentire la rimozione di un sensore dal sistema
    - 10.1 dissociare il sensore dalla zona a cui è eventualmente assegnato
11. consentire l'aggiunta di attuatori al sistema

    11.1 specificare le azioni attuabili dall'attuatore
    11.2 consentire di definire il comportamento dell'attuatore
12. consentire l'assegnazione di un attuatore ad una e una sola zona esistente
    12.1 notificare l'attuatore dell'associazione avvenuta
13. consentire la dissociazione di un attuatore dalla zona cui è assegnato
    13.1 notificare l'attuatore della dissociazione avvenuta
14. consentire l'aggiunta di zone
    14.1 specificare un nome univoco da associare alla zona
15. consentire la rimozione di zone esistenti
    15.1 dissociare i sensori associati dalla zona rimossa
    15.2 dissociare gli attuatori associati dalla zona rimossa
16. esporre lo stato delle zone
    16.1 accedere alle misure aggregate dei sensori appartenenti alla zona
    16.2 accedere alle azioni attualmente in esecuzione
17. consentire la definizione di categorie assegnabili ai sensori
18. consentire la definizione di azioni assegnabili agli attuatori
    18.1 consentire la parametrizzazione di un'azione
19. consentire la visualizzazione delle regole definite
20. consentire la rimozione di un attuatore
21. consentire la visualizzazione delle zone presenti nel sistema

Termine	Significato
Logiche di aggregazione	Ogni zona può avere più sensori dello stesso tipo e i loro valori andranno aggregati prima di poter essere valutati. Il nostro framework, quindi, espone un metodo per definire tali logiche di aggregazione.
Parametri	Un attuatore fisico comprende una serie di azioni specificate da chi usa il nostro framework. Tali azioni possono essere parametrizzate e.g. per l'azione "innaffiare" si può esprimere la potenza di irrigazione.
Stato ottimale	Ogni pianta ha delle condizioni ottimali di crescita (umidità al 70%, per esempio) e l'obiettivo della definizione di regole è di mantenere questo stato ottimale.

Tabella 2: Disambiguazione terminologica

3.4 Non funzionali

Usability	Il framework deve consentire di definire una serra intelligente in maniera semplice
Cross-platform	Il sistema deve funzionare correttamente su diversi sistemi operativi, quali: MacOS, Linux e Windows

3.5 Di implementazione

Il framework sarà implementato utilizzando come linguaggio principale Scala e testato mediante Scalatest, utile a scrivere test in maniera rapida.

4 Design architetturale

A seguito dell'analisi dei requisiti è emersa la necessità di sviluppare un sistema distribuito ed eterogeneo in cui diverse entità possano dialogare. In particolare, il pattern architetturale *client-server* è quello che si sposa meglio con il contesto di questo progetto. Il server rappresenta la serra intelligente che mette a disposizione vari servizi per potervi interagire. Il ruolo di client è ricoperto sia dagli utenti che usufruiscono dei servizi per mezzo di un client remoto, sia dai dispositivi intelligenti (i.e. sensori e attuatori) che contribuiscono ad alimentare il sistema. Nonostante utenti e dispositivi intelligenti possano essere visti entrambi come client, in realtà interagiscono con il server in maniera differente. Nella tradizionale architettura client-server con pattern request-response, infatti, è il client ad effettuare una request verso il server e il viceversa non è possibile. Nel nostro framework:

- l'utente remoto è il vero e proprio client che contatta il server per effettuare delle richieste e ne riceve passivamente le risposte;
- i dispositivi intelligenti, invece, necessitano di un canale bidirezionale dove è possibile sia essere contattati dal server che fornire attivamente risposte.

In figura 3 si riporta l'architettura complessiva del sistema, suddivisa nei principali sotto-moduli identificati, ossia i) server; ii) client; iii) device. Segue il dettaglio delle diverse entità presenti all'interno dei moduli.

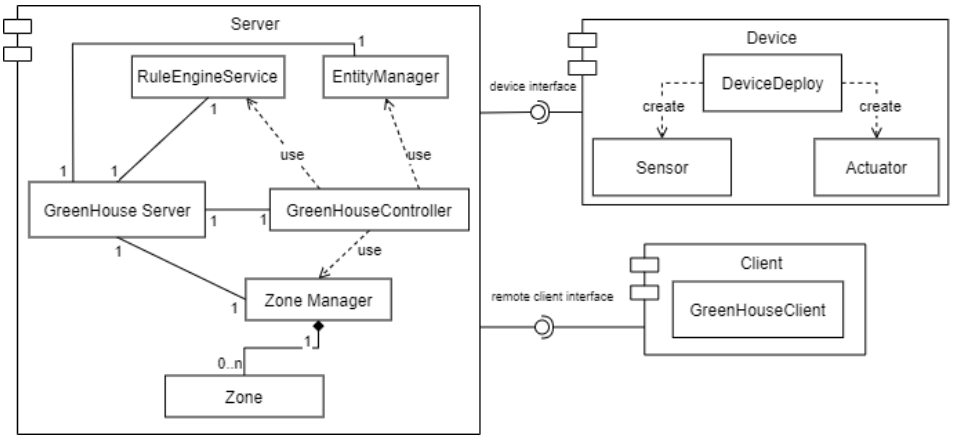

Figura 3: Architettura complessiva del sistema

4.1 Server

GreenHouseServer Questo componente funge da entry point del sistema, infatti è il componente core dell'architettura server e si occupa di creare tutti gli altri componenti: `EntityManager`, `ZoneManager`, `RuleEngineService`, nonché di inoltrare le richieste ricevute dal `Client` al `GreenHouseController`.

ZoneManager Questo componente è stato pensato per rispondere a quanto espresso dal requisito [14.] ed è deputato alla gestione della serra in quanto agglomerato di zone. In particolare:

- tiene traccia delle zone presenti;
- permette di aggiungere e rimuovere zone;
- si occupa dell'associazione di un sensore/ attuatore ad una zona esistente;
- si occupa della dissociazione di un sensore/ attuatore da una zona esistente.

La scelta di deputare allo `ZoneManager` l'associazione delle entità deriva dal fatto che prima di associare un'entità è necessario controllare che questa non sia già assegnata ad altre zone. Le zone, essendo indipendenti, hanno una conoscenza locale in merito alle entità associate e quindi non sarebbero il luogo ideale.

Zone Questo componente gioca un ruolo chiave condensando al suo interno buona parte della logica del framework e interagendo con tante altre componenti parte dell'architettura. Per soddisfare il requisito [16.] è necessario che la zona calcoli periodicamente uno stato. Questo stato interno si compone delle percezioni dei sensori e delle azioni attualmente in esecuzione. Se per le prime lo stato ne riflette una misura aggregata per categoria, calcolata ogni tot tempo, le azioni attive devono essere fruibili in real-time. Sebbene sia la zona stessa a "comandare" le azioni sugli attuatori ad essa associati, la determinazione delle azioni da effettuare verrà demandata ad un componente apposito chiamato `RuleEngineService`.

Riassumendo, gli aspetti cardine che la zona gestisce sono:

- ricezione dei valori prodotti dai sensori ad essa associati;
- ricezione in real-time dello stato degli attuatori ad essa associati;
- aggregazione periodica dei valori prodotti dai sensori di una stessa categoria;
- azionamento degli attuatori, per mezzo della valutazione periodica effettuata dal `RuleEngine` sulla base delle misure aggregate.

EntityManager Questo componente funge da "single source-of-truth" di tutte le entità fisiche che partecipano alla serra, dove per entità si intendono sensori e attuatori. Poiché questi possono partecipare in maniera dinamica, tramite operazioni di `join` e `leave`, vi è la necessita di tenere traccia di quali sono quelli attualmente presenti. `EntityManager`, però, non gestisce l'associazione delle entità alle zone, ma bensì l'ingresso nel sistema di entità assegnabili e la loro eventuale uscita. In linea con il requisito [10.] è chiaro che, qualora un'entità fosse associata ad una zona, la sua rimozione dal sistema attraverso un'operazione di `leave`, ne comporterebbe la dissociazione.

RuleEngineService Il ruolo principale di questo servizio è quello di fornire la funzionalità di valutazione di regole d'attuazione a partire dallo stato di una zona. In particolare, gestisce tutti gli aspetti relativi alle regole di attuazione, infatti:

- permette di aggiungere e rimuovere regole;
- permette di abilitare e disabilitare regole;
- permette di valutare le regole a partire da un insieme di misure rilevate.

GreenHouseController Questo componente ha il ruolo di gestire ogni richiesta di tipo request-response ricevuta dal `Client` verso il server. In particolare riceve e gestisce:

- richieste che riguardano le zone tramite lo `ZoneManager` (e.g. stato di una zona [16.], eliminazione zona [15.]);
- richieste che riguardano le regole tramite il `RuleEngineService` (e.g. abilitazione [4.] e disabilitazione regole [5.]);
- richieste che riguardano sensori e attuatori tramite l'`EntityManager` (e.g. rimozione di entità dal sistema [10. e 20.]).

4.2 Client

Il lato client dell'architettura del nostro framework espone la possibilità di interfacciarsi con il server per effettuare tutte le azioni a disposizione dell'utente finale (e.g. bracciante). In particolare:

- creare [UserStory-10] e rimuovere zone [UserStory-11];
- visualizzare le zone presenti nella serra [UserStory-12] e richiederne lo stato [UserStory-1];
- associare [UserStory-13] e dissociare entità da una zona [UserStory-14];
- rimuovere un'entità dal sistema [UserStory-8, UserStory-9];
- accedere alle regole definite per la serra [UserStory-15];
- abilitare [UserStory-5] e disabilitare [UserStory-4] regole.

4.3 Sensori e Attuatori

I dispositivi intelligenti sono racchiusi nel modulo `device`, dove sono presenti tre entità:

- `Sensor` tramite cui viene rimappato un sensore fisico in un sensore logico nel nostro framework e che si occupa di inviare le percezioni dal mondo fisico al server;

- `Actuator` tramite cui viene rimappato un attuatore fisico in un attuatore logico nel nostro framework e che si occupa di ricevere comandi dal server ripercuotendoli nel mondo fisico;

- `DeviceDeploy` il cui ruolo è quello di effettuare il "join" di `Sensor` e `Actuator` [UserStory-6, UserStory-7].

4.4 Interazioni

Tutte le interazioni all'interno del sistema avverranno tramite scambio di messaggi. In particolare sono state identificate tre macro-interazioni tra le seguenti entità:

- Client e Server: questa interazione si verifica ogni qualvolta il client effettua una richiesta verso il server e sfrutta il pattern request-response, ovvero il client attende una risposta dal server. In questo caso non è stata effettuata una progettazione REST del protocollo per motivi strettamente legati alle ore a disposizione, si è quindi preferito modellare il protocollo solo in termini di messaggi di request-response scambiati;

- Dispositivi e Server: anche questa interazione rispetta il pattern request-response e si verifica quando un dispositivo vuole entrare a far parte del sistema, ovvero quando effettua una operazione di "join". Tale richiesta avrà esito negativo quando effettuata per un dispositivo con un identificativo già registrato. Una volta che il dispositivo viene associato ad una zona, la comunicazione si sposta sul segmento *Dispositivo-Zona*;

- Dispositivi e Zona: una volta che un dispositivo viene associato ad una zona, l'interazione avviene tra il dispositivo e la zona stessa. In questo segmento la comunicazione avviene tramite scambio di messaggi in maniera bi-direzionale. In particolare, il dispositivo comunicherà alla zona eventuali cambiamenti di stato (e.g. attuatori) o nuove misure rilevate (e.g. sensori), mentre la zona comunicherà al dispositivo eventuali comandi remoti. Nel caso degli attuatori, ad esempio, i comandi coincideranno con le azioni da attuare. La dissociazione potrà avvenire una volta che ci si trova in questo segmento di comunicazione e sarà la zona stessa ad informare il dispositivo che non è più associato ad essa.

5 Design di dettaglio

5.1 Premesse

La realizzazione dell'intero framework si baserà sul modello ad attori. L'attore è un'entità che incapsula uno stato, un comportamento e che ha un suo flusso logico. Alcuni degli aspetti fondamentali della semantica definita dal comportamento dell'attore sono: un comportamento reattivo, location transparency e macro-step semantics.

Si è deciso di utilizzare il paradigma ad attori per rispondere efficacemente alla necessità di avere un sistema distribuito [requisiti 1.5 e 2.]. Tale modellazione, infatti, esonera dalla gestione di logiche di concorrenza e aspetti di basso livello che servirebbero a fare comunicare i componenti principali (e.g. `Server`, `Client`, `Actuator` e `Sensor`). Un esempio lampante di concorrenza riguarda l'interazione tra `Zone` e `Sensor` in quanto la prima potrebbe contenere un numero elevato di sensori ed attuatori che le invierebbero un ampio flusso di informazioni anche contemporaneamente. Attraverso il modello ad attori, inoltre, è possibile creare servizi isolati che comunicano tra loro attraverso scambio di messaggi. Ove possibile, l'utilizzo di questa modellazione verrà nascosta a chi estende il nostro framework tramite l'uso di facciate che sfruttano la programmazione asincrona.

5.2 Core concepts

Questa sezione descrive i concetti principali del nostro framework che saranno sfruttati per gestire le logiche di funzionamento della serra intelligente.

5.2.1 Dispositivi e capacità

Il diagramma riportato in figura 4 mostra i principali concetti legati ai dispositivi intelligenti.

Uno dei principali concetti del framework è il `Device`; in particolare, secondo i requisiti 6. e 11. esistono due tipologie di dispositivi intelligenti: i) `Sensor` e ii) `Actuator`, le cui interfacce e modalità d'uso vengono dettagliate nella sezione 5.5 relativa al modulo dei dispositivi. Ad alto livello, ogni `Device` è caratterizzato da un nome (o identificativo) e da una capacità (`Capability`). Quest'ultima rappresenta cosa un dispositivo è in grado di fare (e.g. percepire o attuare). In questo contesto vengono modellate due capacità:

- `SensingCapability`: relativa ai sensori, rappresenta la capacità di un dispositivo di percepire una qualche grandezza fisica dall'ambiente. Tale tipologia di grandezza viene identificata per mezzo di `Category`;
- `ActingCapability`: relativa agli attuatori, descrive l'abilità di un dispositivo di saper attuare una classe di `Action` descritte da un `ActionTag`.

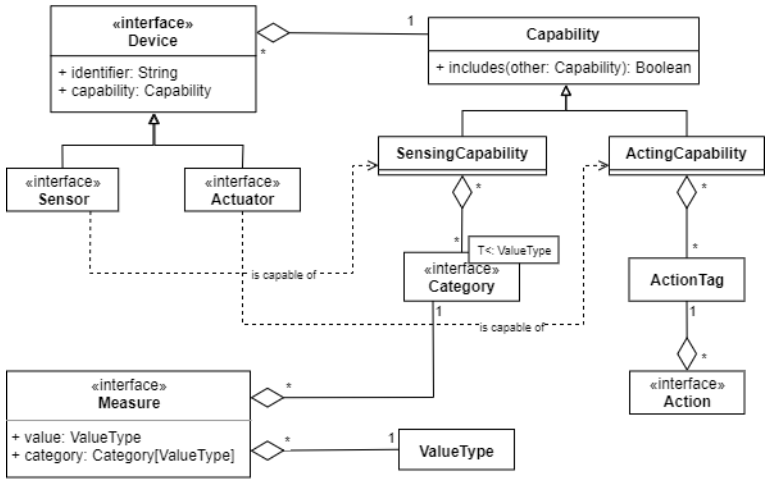

Figura 4: Diagramma delle classi dei principali concetti relativi ai dispositivi intelligenti

Per poter soddisfare il requisito 11.1 vengono introdotti `Action` e `ActionTag`, che benché possano apparire simili, presentano una netta differenza: `Action` rappresenta un comando concreto da inviare ad un attuatore (e.g. dare acqua per 20 secondi), mentre l'`ActionTag` può essere visto come la tipologia del comando (e.g. dare acqua). In questo modo, lo sviluppatore potrà definire arbitrariamente i comandi da inviare agli attuatori, ovvero le `Action` e attribuirgli un `ActionTag`. Quest'ultimo, poiché non rappresenta l'istanza di un comando ma semplicemente una classe di azioni, può essere utilizzato per descrivere delle `ActingCapability` di un certo `Actuator`.

I sensori sono caratterizzati da una `SensingCapability` che a sua volta si compone di una `Category`. Quest'ultima rappresenta la tipologia di grandezza fisica percepita da un sensore, come ad esempio temperatura e umidità. L'intento è quello di consentire allo sviluppatore che utilizza la libreria di definire delle proprie `Category` e associarle ai `Sensor`, secondo quanto richiesto dal requisito 6.1. La definizione di una nuova `Category` richiede allo sviluppatore di effettuare il "bind" del template `T` ad un `ValueType` che rappresenta un tipo primitivo supportato dalla libreria. Ogni sensore, dunque, è in grado di produrre delle `Measure` contenenti un valore concreto di tipo `ValueType` e in relazione ad una `Category` (e.g. temperature 10.3). Maggiori dettagli su `ValueType` sono riportati nella sezione 5.2.3.

5.2.2 Stato operazionale

Il diagramma in figura 5 descrive lo stato operazionale di un attuatore. Il nostro framework, infatti, prevede la presenza di entità (attuatori) in grado di effettuare

determinate `Action` al verificarsi di alcune condizioni. Il requisito 16.2 porta alla reificazione del concetto di `OperationalState` che indica lo stato di un attuatore e in particolare permette di:

- descrivere le azioni che un attuatore sta eseguendo;
- aggiungere nuove azioni a quelle attualmente in esecuzione;
- rimuovere azioni qualora l'attuatore ne avesse terminato l'esecuzione.

Un `OperationalState` risulta `Idle` quando l'attuatore non sta effettuando nessuna azione, viceversa coinciderà con un `DoingActions` composto da un insieme di `Action`.

`Action` descrive il comando che un attuatore può ricevere per eseguire un'azione. `TimedAction` e `ToggledAction` sono le tipologie di azioni gestite.

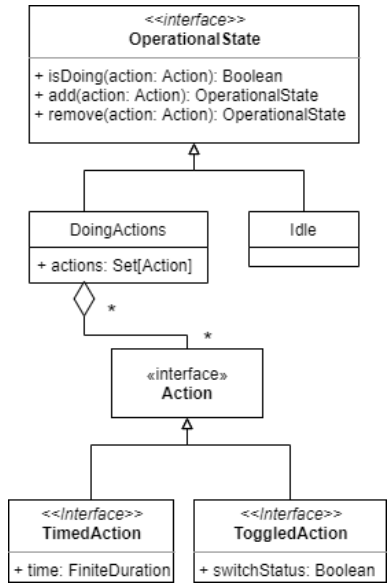

Figura 5: Diagramma delle classi del modulo legato allo stato operazionale di un attuatore

5.2.3 ValueType

La figura 6 mostra il diagramma UML del concetto `ValueType` tramite cui si indicano i tipi ammessi per i valori prodotti dai sensori. La strutturazione da noi imposta serve a semplificare il nostro lavoro di controllo: evitiamo di dover gestire tutti i possibili tipi di dato complessi (tuple ad esempio). Tuttavia, al fine di ottenere un framework estendibile, questi tipi di dato possono comunque essere

rappresentati mediante delle stringhe, demandando la loro serializzazione/ deserializzazione all'utilizzatore del framework. Scala infatti (requisito 3.5), non permette di vincolare questi tipi di dato in modo semplice (e.g. `AnyVal` non comprende le stringhe) e quindi è stata creata questa gerarchia di tipi che fungono da semplici "wrapper" dei tipi "standard".

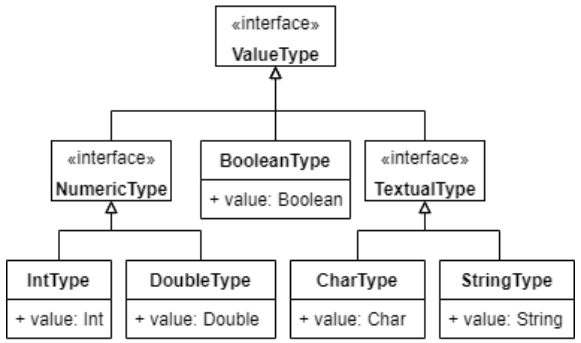

Figura 6: Diagramma delle classi legato al tipo del valore comunicato da un sensore

5.2.4 Stato di una zona

Il concetto di stato nasce dall'esigenza espressa dal requisito 16. e in riferimento ai sotto-requisiti 16.1 e 16.2 si compone delle:

- misure percepite e aggregate dei sensori associati;
- azioni attualmente attive sugli attuatori associati.

Come descritto nel paragrafo 5.3.4, lo stato viene calcolato periodicamente per ogni zona e ne fornisce una visione complessiva. Per tale ragione è stato introdotto anche un `timestamp` che consente di determinare la "freschezza" dello stato, ovvero in quale istante temporale è stato calcolato. Inoltre, essendo `State` una struttura dati immutabile, garantisce di evitare stati non validi o side-effect indesiderati sulla struttura (e.g. cambio del timestamp). La stessa rappresentazione di `State` viene condivisa tra il client e il server. In figura 7 si riporta il diagramma delle classi inerente a `State`.

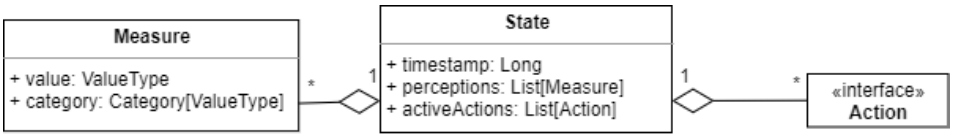

Figura 7: Diagramma delle classi relativo alla struttura dello stato di una zona

5.2.5 Regole

Il concetto di regola è di basilare importanza per una corretta interpretazione dei requisiti 2.1 e 2.2. `Rule` è un'entità che comprende:

- un *antecedente*, identificato da uno o più `ConditionStatement`, che verrà valutato da `RuleEngine` per capire se la serra è in uno stato sub-ottimo;
- un *conseguente* che consiste nell'insieme di `Action` da far eseguire agli attuatori disponibili a fronte del soddisfacimento dell'antecedente.

`RuleEngine` è il componente che si preoccupa di valutare lo stato della serra ricevuto periodicamente e di inferire le eventuali azioni da eseguire per ripristinare uno stato ottimale.

Un `AtomicConditionStatement` rappresenta una singola condizione in una regola ed è caratterizzata da una terna composta da i) una categoria (`Category`), ii) un operatore (`ConditionOperator`), iii) un valore (`ValueType`). Il framework supporta `ConditionOperator` che "wrappano" gli operatori di comparazione tradizionali (e.g. "<", ">"). Un esempio di condizione atomica è *"Temperatura > 35"*. Un `AndConditionStatement` riguarda tanti `ConditionStatement` in *and logico* tra loro.

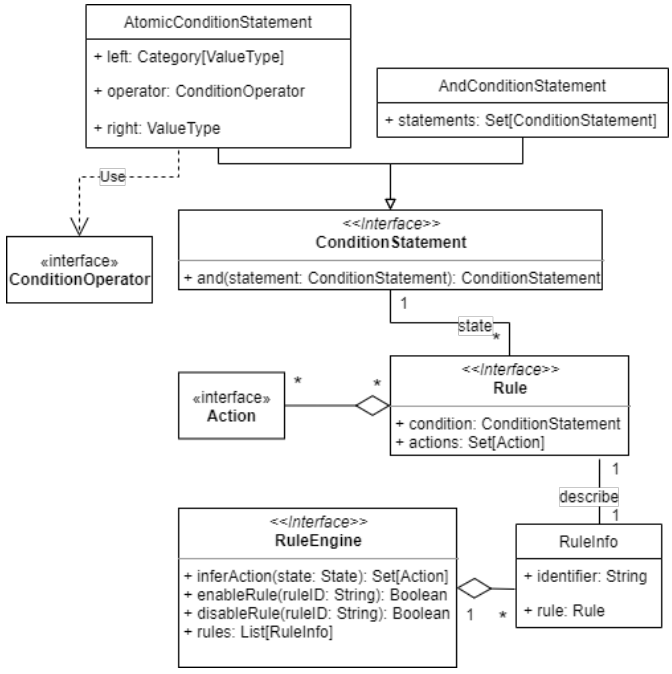

Figura 8: Diagramma delle classi legato alle regole

La struttura delle `Rule` progettata genera una sorta di AST che la rende indipendente dalla modalità di valutazione. Infatti, tale compito sarà demandato al `RuleEngine` la cui implementazione definirà "come" valutare una `Rule`.

5.3 Server

In figura 9 si riporta il diagramma delle classi che illustra i principali componenti del `server`. La maggior parte di questi, fungendo da microservizi, sono stati pensati come attori. `GreenHouseActor`, con il quale è possibile interagire attraverso la facade `GreenHouseServer`, si occuperà dell'avvio e dello spegnimento del server istanziando i servizi di cui si compone e creando il `GreenHouseController`. Quest'ultimo non è altro che un router a cui saranno inoltrati i messaggi provenienti dal client, in modo da inglobare al suo interno la logica di redistribuzione tra i vari microservizi. Come si può notare in figura 9 i servizi esposti dal server si celano dietro `EntityManagerActor`, `ZoneManagerActor` e `RuleEngineService`, il cui comportamento di massima è stato descritto in fase di design architetturale. Di seguito vengono descritti alcuni dettagli dei singoli servizi:

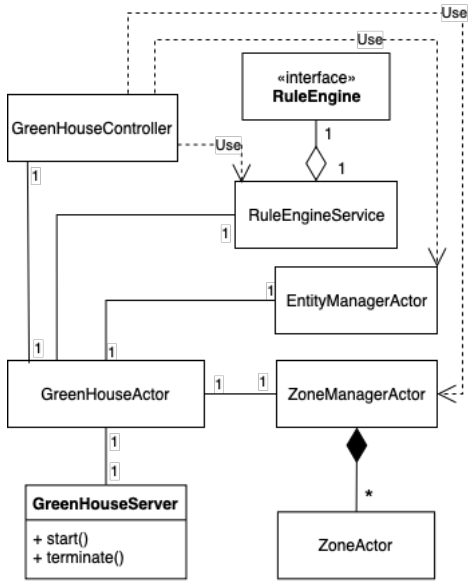

Figura 9: Diagramma delle classi del modulo server

5.3.1 EntityManagerActor

`EntityManagerActor` è incaricato di gestire, attraverso la ricezione di messaggi, l'ingresso e l'uscita delle entità dal sistema. Dal momento in cui la partecipazione al sistema come device non implica l'associazione ad una zona, queste

due logiche sono gestite in punti separati. Tuttavia, al fine di mantenere coerenza nello `ZoneManagerActor`, che si occupa di mantenere le associazioni dei device alle diverse zone, è bene che quando un'entità viene rimossa attraverso l'`EntityManagerActor`, questo lo notifichi allo `ZoneManagerActor`, in modo da poter rimuovere l'associazione. In questa fase si è quindi evidenziata la necessità di una comunicazione tra le due parti e, per mantenere le entità disaccoppiate si è preferito un pattern di interazione di tipo publish-subscribe, rispetto a una composizione. Lo `ZoneManagerActor`, quindi, si registrerà ad un topic per poter ricevere notifiche relative alla rimozione di entità dal sistema.

5.3.2 RuleEngineService

`RuleEngineService` è il servizio deputato alla ricezione di messaggi inerenti alle regole, al fine di abilitarle/disabilitarle e inferire le azioni da intraprendere. Anch'esso rappresentato per mezzo di un attore, funge principalmente da punto di accesso alle logiche che coinvolgono le regole, "wrappando" un'istanza di `RuleEngine`, all'interno della quale vengono mantenute le logiche applicative.

5.3.3 ZoneManagerActor

`ZoneManagerActor` è il servizio che, sempre per mezzo di un attore, gestisce i messaggi per creare/ rimuovere zone e associarvi/ dissociarvi entità. A fronte del requisito 14.1 la richiesta di creazione di una zona prevederà l'inserimento di un nome univoco da associarvi con conseguente "spawn" di un attore.

5.3.4 ZoneActor

La zona, il cui diagramma delle classi è espresso attraverso la figura 10, è un concetto predominante ai fini del funzionamento del sistema ed è stata progettata come un attore principalmente per due motivi:

- facilitare la gestione di aspetti di concorrenza derivanti dai diversi flussi di controllo che interagiscono con essa; in modo particolare i device, per la pubblicazione delle misure e per l'aggiornamento delle azioni attive;
- sfruttare la reattività al tempo dell'attore, al fine di gestire sia il calcolo periodico dello stato sia delle azioni da intraprendere per ripristinare le condizioni ideali della serra.

Segue la descrizione dei macro comportamenti della zona:

Comunicazione con le entità La zona deve poter comunicare con le entità ad essa associate. A tale scopo si utilizza l'astrazione `DeviceChannel` che contiene il riferimento al `Device` ed espone un modo per contattarlo.

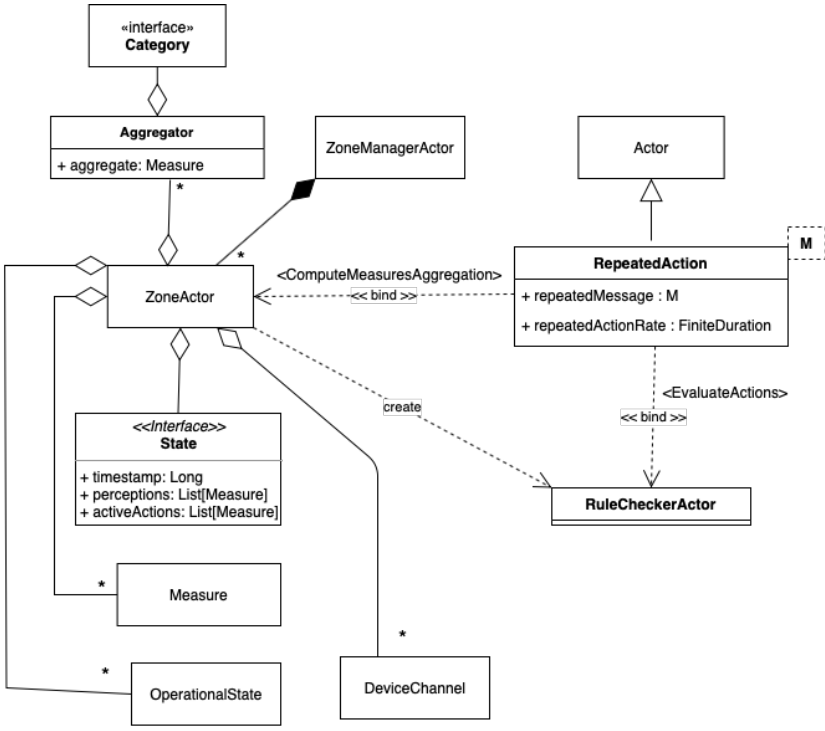

Figura 10: Diagramma della zona

Collezionamento dei valori dei device Nell'arco del ciclo di vita della zona verranno continuamente collezionati i cambiamenti di stato degli attuatori e le misurazioni ricevute dai sensori. Dal momento in cui ogni sensore ha un suo periodo di inoltro della misurazione effettuata, può essere che questo la invii più volte all'interno di una stessa finestra di computazione delle misure aggregate. Per questo motivo si è deciso di mantenere la misura più recente prodotta da uno stesso sensore. Lo stato interno alla zona, esposto attraverso un messaggio, viene computato ogniqualvolta un attuatore cambia il suo stato (inizia/termina l'esecuzione di un'azione) o avviene l'aggregazione delle misure.

Reazione al tempo Periodicamente, potenzialmente in tempi distinti, sarà compito della zona calcolare lo stato aggregato e chiedere al `RuleEngineService` di inferire le azioni da attivare sulla base dello stato precedentemente computato. `RepeatedAction` è un concetto che abilita un attore a ricevere messaggi periodici. Chiunque estenda questo concetto non dovrà far altro che specificare il periodo di scheduling e il messaggio che intenderà gestire per innescare l'invio periodico del messaggio definito. Per rispettare il principio del Single Responsibility Principle, essendo il tempo di calcolo dello stato diverso da quello di inferenza delle azio-

ni, si è preferito demandare quest'ultimo compito ad un'entità a parte chiamata `RuleCheckerActor`. Questo farà da ponte tra `ZoneActor` e `RuleEngineService`, inoltrando le azioni inferite alla zona, in modo che questa possa redirezionarle agli attuatori competenti in linea con le loro `Capability` (requisito 1.2). Per quanto riguarda il calcolo dello stato aggregato è necessario che la zona si avvalga di una serie di aggregatori, ognuno capace di calcolare una misura aggregata a partire da misure di una stessa categoria. Ogni aggregatore si riferisce quindi ad una specifica `Category`, che ingloba al suo interno. Ogni zona deve avere un unico modo di aggregare le misure di una certa `Category`. In fase di implementazione si dovrà quindi impedire l'associazione di aggregatori diversi per una stessa categoria alla stessa zona. Per riassumere quanto detto si riportano due diagrammi di sequenza che dettagliano le interazioni tra gli attori per il calcolo dello stato e per le azioni inferite.

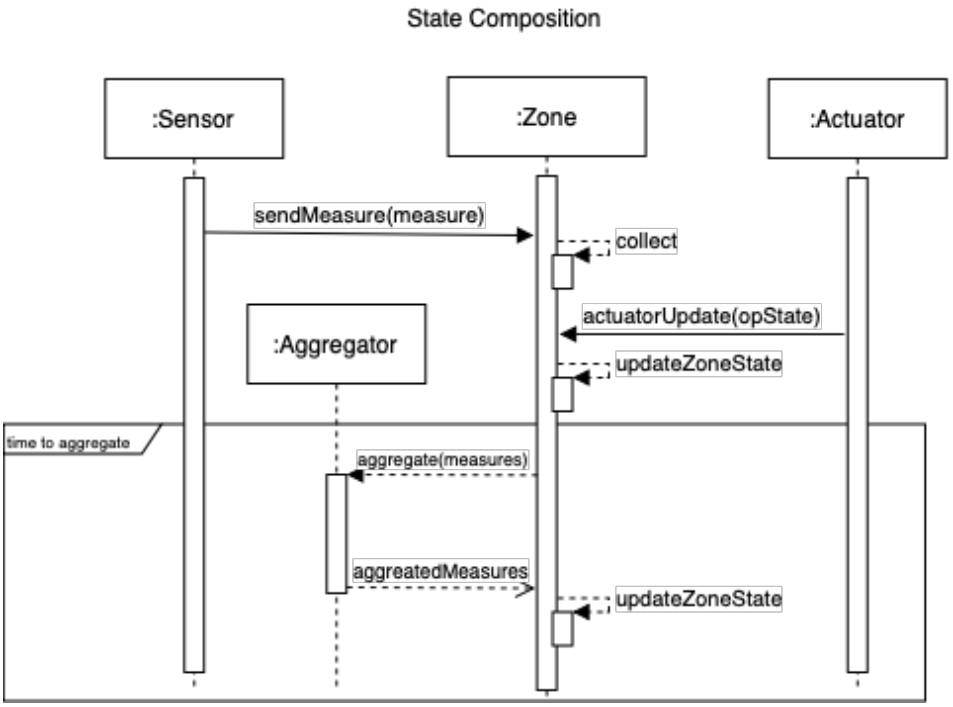

Figura 11: Diagramma di sequenza inerente all'interazione tra gli attori per il calcolo dello stato

5.4 Client

In figura 13 viene riportato il diagramma delle classi che dettaglia il modulo relativo al `client`, comprensivo delle principali funzionalità.

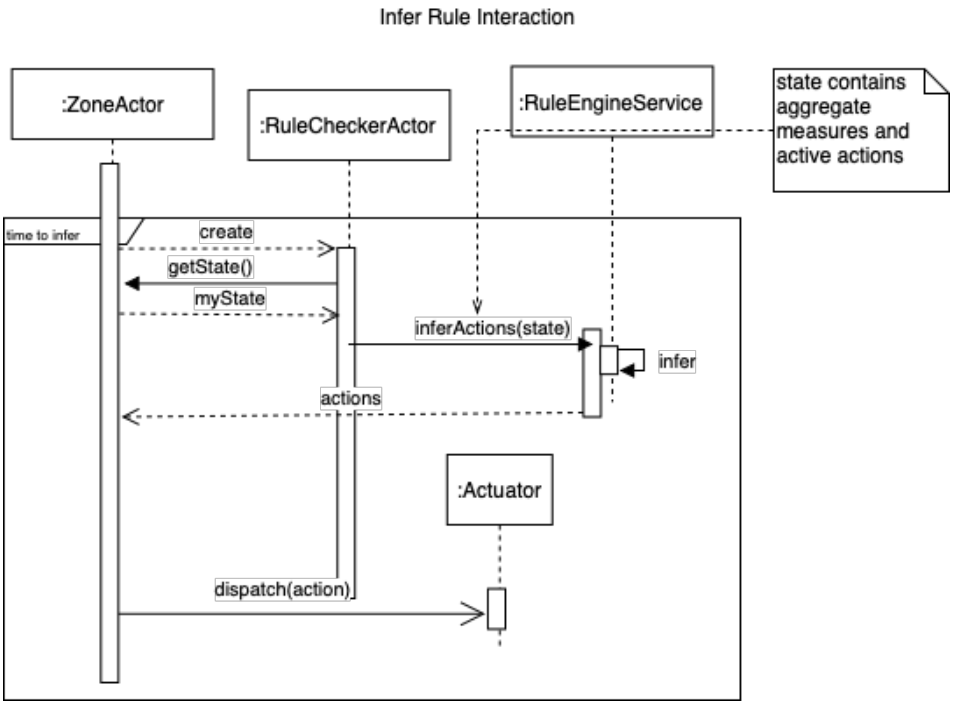

Figura 12: Diagramma di sequenza inerente al calcolo delle azioni da eseguire sugli attuatori

Questo modulo si occupa di gestire le richieste dell'utente comunicandole al server.

Le principali funzionalità del client sono raccolte nelle due interfacce: i) `RuleClient` (secondo i requisiti 4., 5. e 19.) e ii) `ZoneClient` (secondo i requisiti 7., 8., 12., 13., 14., 15., 16. e 21.). La realizzazione di entrambe le interfacce è `GreenHouseClient` che aggiunge le funzionalità specificate nei requisiti 10. e 20.).

In particolare, `GreenHouseClient` è una *facade* utilizzata per esporre tutte le funzionalità del client e per nascondere all'utilizzatore il modello ad attori sottostante sfruttando la programmazione asincrona. La logica per effettuare le richieste al server sarà contenuta nell'attore `Client` che utilizza lo scambio di messaggi per costruire un'interazione di tipo request-response.

5.5 Device

In figura 14 si riporta il diagramma delle classi che descrive in maniera più dettagliata il modulo relativo ai `device` e le principali funzionalità.

Come descritto in architettura 4.3, la classe `DeviceDeploy` consente di effettuare l'operazione di "join" nel sistema. In particolare questa classe espone la

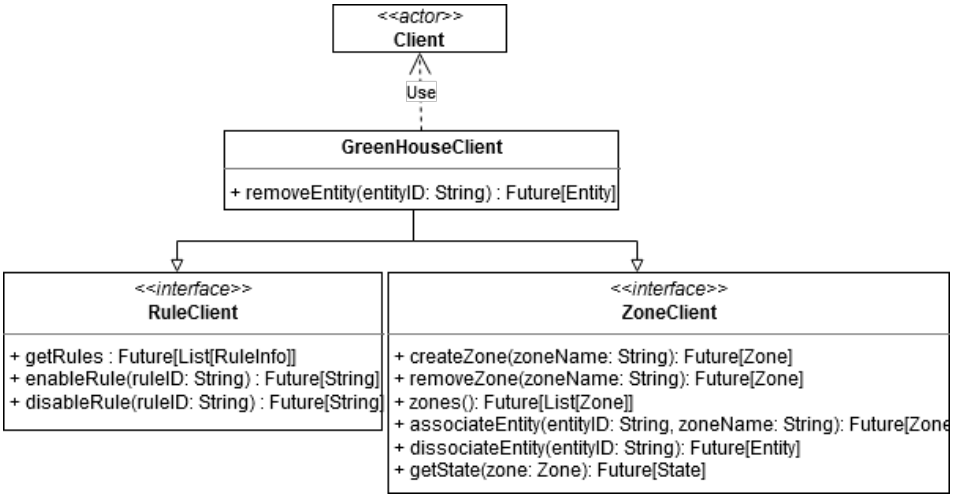

Figura 13: Diagramma delle classi del client

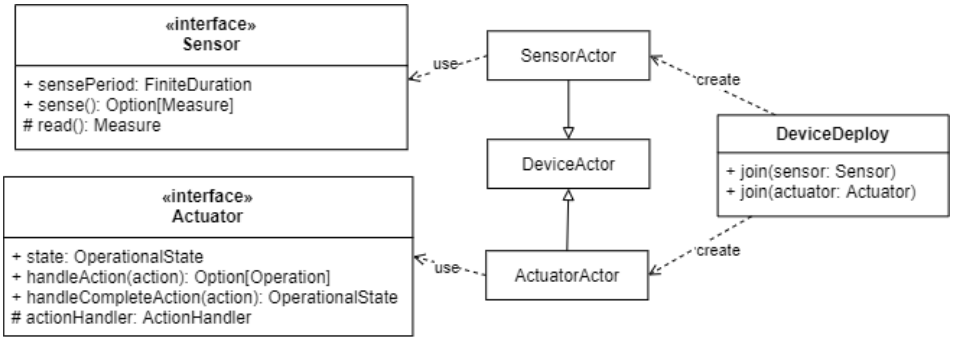

Figura 14: Diagramma delle classi dei dispositivi

possibilità di effettuare un "join" di un sensore o di un attuatore e per farlo crea rispettivamente un `SensorActor` e un `ActuatorActor`. Quest'ultimi sono attori ed utilizzano le interfacce `Sensor` e `Actuator`, le cui implementazioni concrete sono lasciate all'utilizzatore del framework (6.2 e 11.2). `DeviceActor` definisce il comportamento base dei dispositivi intelligenti, ovvero gestisce il segmento di comunicazione tra `Zone` e sensori/ attuatori.

L'implementazione dell'interfaccia `Sensor` viene demandata all'utilizzatore della libreria, che può definire:

- le `SensingCapability`, specificando quale categoria il sensore riesce a percepire, come da requisito 6.1;
- il comportamento del sensore, specificando come produrre una misura e il

periodo di lettura del sensore, come da requisito 6.2.

Discorso analogo vale per `Actuator`, come da requisiti 11.1 e 11.2. L'utilizzatore del framework ha il compito di definire, oltre alle `ActingCapability`, un `ActionHandler`: una funzione che definisce, a fronte dello stato attuale e della ricezione di una `Action`, quale azione fisica intraprendere (e.g. al comando "dare acqua" aziono un'elettrovalvola). Poiché alcune operazioni sul mondo fisico possono essere legate al tempo, o in generale possono cambiare lo stato dell'attuatore in maniera asincrona, viene introdotto il concetto di `Operation`. Ad esempio, l'azione "innaffiare per 20 secondi" implica che lo stato dell'attuatore per 20 secondi sia impostato su "sto innaffiando"‘ e al termine torna in "idle". La libreria tramite `Operation` vuole modellare proprio questo aspetto: specificare se un'operazione è stata completata immediatamente o verrà completata in futuro.

6 Implementazione

La fase implementativa è iniziata con la scelta delle tecnologie specifiche da utilizzare.

In particolare, per implementare il modello ad attori è stato scelto il framework *Akka*[2] nella sua interpretazione classica: il team non ha particolare esperienza con *Akka Typed* e ha deciso di non investire le poche ore a disposizione nel suo approfondimento. La configurazione scelta per gli attori sfrutta *Akka remote* che permette di ottenere un `ActorSystem` in grado di gestire attori remoti. *Akka cluster*, sebbene sia più completo e potente, è stato scartato in quanto offre funzionalità non necessarie alla realizzazione del nostro framework e complicherebbe inutilmente la strutturazione.

Ulteriore scelta implementativa è quella legata alla gestione del `RuleEngine` che è stata effettuata sfruttando *Prolog* nella sua implementazione *tuProlog*[3]. La scelta di utilizzare la programmazione logica all'interno del progetto deriva dalla volontà del team di esplorare quanti più paradigmi di programmazione possibile.

Il team, prima di procedere con la suddivisione dei task e la loro implementazione, ha effettuato una fase di setup del progetto e organizzazione del codice in cui sono stati individuati i seguenti package principali:

- *client*, che contiene tutte le entità relative al *Client*;
- *common*, che contiene tutti i concetti in comune tra i diversi package. Nello specifico sono presenti la configurazione di *Akka* e i protocolli di comunicazione tra i diversi attori, definiti entrambi nelle fasi implementative iniziali;
- *core*, che contiene l'implementazione di tutti i concetti identificati come core già in fase di design. Le interfacce di questi concetti sono state implementate dall'intero team nelle fasi implementative iniziali;
- *device*, che contiene tutti i concetti relativi a sensori e attuatori;
- *server*, che riguarda tutte le entità utili alla realizzazione del lato *Server* del framework.

Il modello di sviluppo software di riferimento per questo progetto è quello del TDD, che prevede: i) la scrittura dei test, ii) il fallimento degli stessi, iii) l'implementazione delle funzionalità per far passare il test iv) una fase di refactoring. Utilizzando questo approccio è stata coperta la maggior parte delle funzionalità implementate.

[2] https://doc.akka.io/docs/akka/current/actors.html
[3] http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome

6.1 Testing

All'interno del progetto, il testing delle varie componenti è avvenuto attraverso l'uso di due principali tool: ScalaTest e AkkaTest.
ScalaTest è stato utilizzato per testare tutte le funzionalità del nostro sistema, sia quelle che includevano gli attori, sia per testare alcune parti specifiche relative al loro comportamento. Per ciascuna classe sono stati realizzati degli unit test che, in maniera graduale, coprissero più casistiche possibili.
Il testing degli attori ha richiesto un approccio differente, in quanto il paradigma di interazione tra e con essi si basa sullo scambio di messaggi. Per testare gli attori abbiamo deciso di crearne alcuni "mocked" e di utilizzarli per seguire il flusso che l'attore sotto test dovrebbe percorrere durante il suo ciclo di vita. In tal modo è possibile controllare che esso risponda correttamente alla ricezione dei messaggi attesi.
AkkaTest viene fornito con un modulo dedicato (`akka-testkit`), utile a supportare i test e utilizzato per tutta quella porzione di progetto che si avvale dell'uso del modello ad attori. In alcuni scenari di test, ove fosse necessario testare i cambiamenti dello stato interno dell'attore, è stato sfruttato il `TestActorRef`. In questo modo è stato possibile controllare il cambiamento a fronte della ricezione di alcuni messaggi.
Alcuni package come `examples` e `intelliserrademo` non sono stati testati poiché contengono codice a scopo dimostrativo.

6.2 Suddivisione del lavoro

La suddivisione del lavoro è stata fatta cercando di assegnare un carico equo ai diversi membri, ma alcune parti si sono dimostrate essere più pesanti di quanto inizialmente previsto. Ad ogni fine sprint sono stati dettagliati e assegnati i task da eseguire nello sprint successivo.

Le seguenti sezioni descrivono quanto implementato mediante tecnica di pair programming e il dettaglio di quanto implementato dai singoli componenti.

6.2.1 Parti in comune

Questa sezione contiene le parti realizzate in stretta collaborazione e, per alcuni concetti, sviluppati tramite la tecnica del pair programming.

RuleEngine in Prolog In primo luogo abbiamo cercato di definire una teoria Prolog che si prestasse alla valutazione di regole di attuazione. Per farlo è stato necessario determinare quali *termini* Prolog fossero idonei a rappresentare i vari concetti di:

- `Action`: la cui rappresentazione è definita dal seguente *compound-term* `action(X)`, dove la variabile `X` corrisponde alla rappresentazione dell'azione da eseguire. Ad esempio `action(water)` indica l'azione "innaffiare";

- `Measure`: viene rappresentata dal seguente *compound-term* `measure(X, Y)`. La variabile `X` corrisponde alla `Category`, mentre `Y` al valore della misura (`ValueType`). Ad esempio `measure(20.4, temperature)` rappresenta una misura di temperatura con valore 20.4;

- `Rule`: viene rappresentata da una regola Prolog, la cui testa indica un `Action` e il corpo contiene una serie di termini *non-ground* di *measure*. Ad esempio `action(water):- measure(X, humidity), X < 40.5.` indica di innaffiare quando l'umidità è inferiore ad un certo valore.

Sono state modellate inoltre le stringhe e i relativi operatori di comparazione per una corretta interpretazione delle `Measure` nel caso in cui il valore fosse di tipo stringa. La logica principale di inferenza risiede all'interno della funzione `infer` scritta in Prolog, che riceve in input una lista di termini *measure* e restituisce una lista di termini *action*.

La teoria di base realizzata in Prolog è presente nel file *greenhouse-theory.pl* all'interno delle *resources* del progetto.

Dovendo passare dalla definizione delle regole in Scala alla sua rappresentazione in Prolog, si è sfruttata la combinazione di Type Classes e Type Enrichment per la conversione di `Rule` in `Term` (concetto della libreria TuProlog). In particolare, la realizzazione del trait `PrologRepresentation` è volta alla definizione di un concetto generico per la traduzione di un oggetto di tipo `T` in termine Prolog. Il listato 1 riporta il trait in questione. `PrologRepresentation` è stato implementato in pair programming per la rappresentazione dei concetti core. Nello specifico, i concetti legati a `State`, `Measure`, `Category` e `ValueType` sono stati realizzati dal gruppo formato da Battistini e Petreti, mentre `Rule`, `ConditionStatement` e `Action` sono stati implementati da Letizi e Luffarelli.

L'uso del pattern *Pimp My Library* abilita l'invocazione diretta del metodo `toTerm` su qualunque oggetto nel cui contesto è definita una `PrologRepresentation` implicita, come riportato nel frammento di codice in 2.

```
trait PrologRepresentation[T] {
  def toTerm(data: T): Term
}
```

Listing 1: PrologRepresentation

```
implicit class RichAny[A : PrologRepresentation](
   toEnrich : A) {
    def toTerm : Term = {
      implicitly[PrologRepresentation[A]].toTerm(
         toEnrich)
    }
}
```

Listing 2: Type Enrichment

La gestione dell'abilitazione/ disabilitazione delle regole in `RuleEngine` è stata realizzata sfruttando i meccanismi di *assert* e *retract* già presenti in Prolog.

Demo La Demo è stata realizzata con l'obiettivo di proporre un esempio di come potrebbe avvenire l'estensione del nostro framework per concretizzare la propria versione di una serra intelligente di pomodori. Nella realizzazione del prototipo il team si è suddiviso in gruppi:

- Luffarelli e Battistini si sono occupate della realizzazione di un'interfaccia grafica in *Scala Swing* sfruttando il client messo a disposizione dal framework;
- Letizi si è occupato della stesura delle regole di attuazione per una serra di pomodori, della definizione degli aggregatori da utilizzare e del deploy della serra intelligente. Inoltre, ha realizzato anche degli attuatori fittizi che potrebbero essere presenti in una reale serra;
- Petreti ha realizzato dei sensori che simulano un andamento sinusoidale dei valori percepiti. L'intento non è quello di ottenere una simulazione realistica, ma di avere un andamento crescente e decrescente che ha poi permesso di verificare l'attivazione di regole di attuazione.

6.2.2 Battistini Ylenia

Nelle prime fasi del progetto ho contribuito maggiormente al modulo relativo al `GreenHouseController`, comprensivo della parte di test, il quale ha il compito di smistare tutti i messaggi in arrivo dal client verso l'attore di competenza. Questa classe può essere vista come una sorta di router che ridistribuisce le richieste ricevute alle entità sottostanti facendo da ponte tra esse. All'interno di `GreenHouseController` troviamo una *receive* la quale riceve tutti i messaggi dal client e, sulla base della richiesta ottenuta invia un messaggio al server. La particolarità di questa classe riguarda il fatto che, quando viene mandato un messaggio al server, `GrenHouseController` specifica quale sarà il sender a cui quest'ultimo dovrà rispondere.
Mi sono poi concentrata sull'implementare, lato `GreenHouseClient` e conseguentemente `Client`, la logica per poter effettuare le richieste di ottenimento di tutte le zone e di tutte le regole presenti nel sistema, con la possibilità di abilitarne una, se precedentemente disabilitata e viceversa.
Un'altra parte da me implementata riguarda la classe `DeviceDeploy` la quale consente di effettuare l'operazione di "join" nel sistema. Per la realizzazione di questa classe ho seguito quanto accordato in fase di architettura, in particolare sono stati implementati, cercando di sfruttare al massimo il paradigma funzionale, due metodi esponendo la possibilità di effettuare il "join" di un sensore o di un attuatore. Entrambi questi metodi confluiscono in una ask verso la classe `EntityManagerActor` la quale, a fronte della seguente richiesta, si occupa effettivamente dell'inserimento del device nel sistema e successivamente di rispondere al sender di quest'ultima. `DeviceDeploy` mappa la risposta ricevuta in una futura di

successo o di fallimento. Se si ricade in quest'ultimo caso viene stoppato l'attore associato all'entità di cui si voleva tentare l'inserimento. Lato client non è possibile inserire nel sistema delle nuove entità; per poter aggiungere sensori ed attuatori, è necessario che l'utilizzatore del framework le definisca ed in un secondo momento le inserisca nel sistema.
Durante l'ultima fase di implementazione mi sono occupata delle classi `RuleEngine` e `RuleEngineService`. Queste sono le classi che si occupano di gestire la comunicazione con il motore Prolog; in particolare `RuleEngineService` realizzato come attore è il servizio deputato alla ricezione di messaggi inerenti alle regole mentre `RuleEngine` mantiene le logiche applicative. All'interno di queste classi ho implementato i metodi e le logiche per la gestione delle regole tra cui la visualizzazione di tutte le regole presenti nel sistema, la possibilità per l'utente finale di abilitare o disabilitare regole precedentemente inserite e, la capacità, fornito in input lo stato di una zona, di inferire l'azione da eseguire. Come per sensori ed attuatori non è possibile per il bracciante inserire nuove regole; l'inserimento di queste risulta possibile solamente tramite le seguenti operazioni: i) stop del server, ii) inserimento di nuove regole, iii) ricaricamento delle regole sul motore Prolog, iv) start del server. Ovviamente l'intero passaggio è a disposizione dell'utilizzatore del framework.

6.2.3 Letizi Simone

Di seguito vengono descritti i macro concetti da me implementati mediante trait o classi, in accordo con quanto espresso in fase di design di dettaglio. Per ognuno di essi è stato mio onere la realizzazione dei relativi test.

Aggregator nell'implementazione dell'aggregatore, la definizione di una factory con argomenti carried ha permesso l'inferenza a compile time tra la categoria e il tipo della funzione di aggregazione. Dal momento in cui la `Category` è generica nel tipo di valore prodotto (`ValueType`), l'utilizzatore del framework viene costretto a definire una funzione di aggregazione tipata coerentemente con la `Category` passata. Il mancato utilizzo del currying consentirebbe, infatti, la definizione di aggregatori con una funzione definita su un ValueType qualsiasi, anche non coerente con quello della Category indicata, e l'errore verrebbe riportato a run-time.

AggregateFunction con lo scopo di semplificare la definizione di aggregatori, è stato implementato l'object `AggregationFunction` che ingloba alcune funzioni di aggregazione di default quali *sum*, *min*, *max*, *avg*, *moreFrequent* e *lessFrequent*. Ovviamente le prime quattro vengono rese fruibili solamente per quelle `Category` che producono un `NumericType` e la loro realizzazione è stata fatta sfruttando le funzioni *sum*, *min* e *max* definite da Scala per i `TraversableOnce`. Queste tuttavia prendono un argomento implicito della type class `Fractional` o `Ordering`, utili a definire le operazioni matematiche e la strategia di ordinamento. A tal proposito è stata necessaria l'implementazione di oggetti impliciti che definiscano le diverse implementazioni di `Fractional` per i `NumericType` (`DoubleType` e `IntType`). Per

quanto riguarda la funzione *avg*, non presente nei `Traversable` di Scala, essa è stata abilitata mediante il pattern *Pimp My Library*, grazie al quale è stato possibile arricchire le collection con alcuni metodi.

Conversioni implicite per ValueType la realizzazione dei tipi wrapped, pur fornendo i vantaggi espressi nella sezione 5.2.3, ha reso il loro utilizzo verboso e poco intuitivo. Trattandosi di un framework, al fine di fornire all'utilizzatore un modo semplice e lineare per definire l'aggregazione dei valori prodotti dai sensori, si sono sfruttate le conversioni implicite dai tipi primitivi del linguaggio ai tipi wrapped. In virtù di ciò, l'utilizzatore del framework maneggerà i tipi primitivi come se i tipi wrapped non esistessero.

Il listato 3 è volto a mostrare l'evidente diminuzione di sforzo che, grazie alla definizione delle funzioni di aggregazione e delle conversioni implicite sui `ValueType`, viene richiesto all'utilizzatore del framework nella creazione degli aggregatori. La riga di codice che non utilizza la funzione di aggregazione `sum` mostra quale sarebbe stato lo sforzo necessario richiesto all'utilizzatore del framework prima di tali migliorie.

```
val tempSumAggregator = createAggregator(Temperature)(
    sum)
val tempSumAggregator2 = createAggregator(Temperature)(
    list => list.fold[IntType](IntType(0))((a1, a2) => a1
    .value + a2.value))
```

Listing 3: Vantaggi introdotti dalla definizione di funzioni di aggregazione

State in accordo con quanto espresso dalle interfacce stabilite in fase di design, si è proceduto all'implementazione dello stato come entità immutabile.

ZoneActor essendo questo attore denso di logiche complesse quali, ad esempio, l'inoltro ai corretti attuatori delle azioni da svolgere, l'uso di costrutti come il for comprehensions di Scala ha permesso la scrittura di codice più leggibile rispetto a quanto sarebbe stato utilizzando filter, flatMap, ecc..

RepeatedAction l'uso dei timer di *Akka* ha facilitato la creazione di un attore che, facendo uso di template method, demanda alle classi che lo estendono la definizione del messaggio periodico che intendono ricevere e il tempo di scheduling.

Join di sensori e attuatori in Entity Manager la gestione delle richieste di join da parte dei device è stata una delle prime cose di cui mi sono occupato. Questa parte, tuttavia, non è risultata particolarmente complessa in quanto prevedeva la semplice memorizzazione delle entità, previa verifica di preesistenza.

Generator/Sample in fase di Demo e testing è emersa la necessità di generare dei valori psuedo-randomici; a tale scopo, la realizzazione dell'object `Generator` è volta a creare una sorta di classe di utility che racchiuda diverse modalità di generazione (un elemento, una lista di elementi, uno stream ecc.) di oggetti specificandone il tipo attraverso l'uso dei generici. Per ogni tipo `T` per il quale si intenda definire una strategia di generazione basterà definire un'implementazione di `Sample`, trait generico composto da un unico metodo *sample* che ingloba la logica di generazione. L'uso di `Sample` e `Generator` è infatti combinato; i metodi dediti alla generazione forniti da `Generator` sono generici in T e vanno alla ricerca di una definizione implicita di `Sample[T]`.

6.2.4 Luffarelli Marta

Il mio contributo implementativo è principalmente relativo al concetto di `ZoneManagerActor` (con relativi test). Più nel dettaglio, ho affrontato il problema di come implementare il pattern Publish-Subscribe tra `EntityManagerActor` e `ZoneManagerActor` in modo da informare quest'ultimo quando un'entità viene rimossa dal sistema. Una possibilità era quella di usare *Distributed Publish-Subscribe*, meccanismo offerto da Akka ma nella configurazione "Cluster". La soluzione da me implementata sfrutta invece l'`EventBus`[4] fornito da Akka nella configurazione "Remote", utilizzata nel nostro progetto. L'implementazione di tale feature ha coinvolto:

- l'`EntityManagerActor`, in cui ho gestito la pubblicazione sull'`EMEventBus` a fronte della rimozione di un'entità;
- il `GreenHouseActor`, dove ho gestito l'iscrizione dello `ZoneManagerActor` al topic giusto;
- lo `ZoneManagerActor` in cui ho effettuato le operazioni di controllo necessarie al momento della pubblicazione di un messaggio sul topic dedicato alla rimozione.

La difficoltà maggiore nell'implementare lo `ZoneManagerActor` è relativa all'organizzazione e alla manutenzione delle varie strutture dati necessarie per una corretta gestione delle zone e delle entità ad esse associate. In particolare, è emersa l'esigenza di processare le richieste di associazione di entità in maniera peculiare: l'interazione inizialmente concepita prevedeva una serie di `Ask` in cascata dal `Client` fino all'entità che si voleva associare (lo `ZoneManagerActor`, infatti, dopo aver controllato la legittimità della richiesta, deve informare la zona e l'entità di questa associazione). Questo iter risultava lungo e pesante e costringeva il `Client` ad attendere che tutte le entità (anche quelle remote come i sensori e gli attuatori) venissero informate e che rispondessero.

Il diagramma UML in figura 15 descrive l'interazione rielaborata, da cui si evince che il `Client` viene informato solamente della legittimità della sua richiesta, non dell'effettiva associazione.

[4] https://doc.akka.io/docs/akka/current/event-bus.html

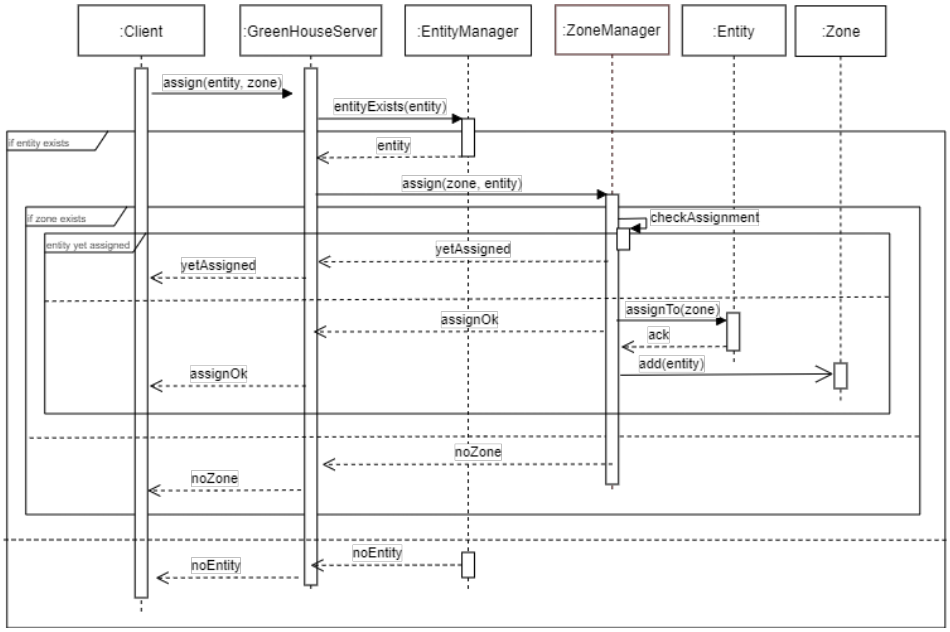

Figura 15: Diagramma di sequenza dell'associazione di un'entità a una zona

Altro concetto da me concretizzato è quello del `ServerConfig` tramite cui definire tutte le configurazioni necessarie alle diverse entità lato server, a partire da host, porta e nome dell'applicazione fino ad arrivare a quelle più specifiche legate alla zona (periodi di campionamento e lista di aggregatori) e alle regole (lista di regole iniziali da specificare). La definizione di queste configurazioni di default ha portato a un refactoring dello `ZoneManagerActor`: prende una *Strategy* all'atto della sua creazione in cui viene specificata la logica di creazione delle zone. Questa scelta ha reso più estendibile tale classe nonché più semplice e pulito testare lo `ZoneManagerActor`.

All'interno del modulo del client e in `GreeHouseActor` nel server ho provveduto a realizzare e testare alcune delle interazioni i.e. associazione/ dissociazione entità, creazione zona, rimozione entità, cancellazione zona.

6.2.5 Petreti Andrea

Nelle prime fasi del progetto ho contribuito maggiormente al modulo relativo al server. Come primo compito ho impostato la configurazione per *Akka Remote* all'interno di `GreenHouseConfig` per poter creare degli *ActorSystem* conformi alla nostra architettura client-server. Mi sono poi dedicato alla realizzazione della *facade* lato server, ovvero ho implementato il trait `GreenHouseServer` e una versione iniziale di `GreenHouseActor`. In particolare, ho cercato di sfruttare l'*ActorSystem* per poter eseguire il server specificando *host* e *porta*. Ho aggiunto inoltre la possi-

bilità di specificare un nome per la serra intelligente che si sta realizzando (sostanzialmente il nome del server). Ho realizzato i relativi test: `GreenHouseServerSpec` e `GreenHouseActorSpec`, approfondendo da subito sia *Scalatest* che *Akkatest*.

Mi sono poi dedicato ad uno dei concetti core del sistema, ovvero le regole di attuazione, implementando il concetto di `Rule`, `ConditionStatement`, `AtomicConditionStatement`, `AndConditionStatement` e `ConditionOperator`. Inoltre, al fine di rendere più semplice per l'utilizzatore comporre le regole, ho realizzato un *dsl* che permette di definire una `Rule` utilizzando la classica simbologia degli operatori logici e di comparazione. Ho cercato di sfruttare alcuni dei punti di forza di *Scala* come gli operatori infissi e gli impliciti per decorare i seguenti concetti:

- `Category` viene decorato con `ConditionCategoryOps`, che espone metodi definiti mediante la simbologia degli operatori di comparazione (e.g. ">=", "<", ecc...). Inoltre, per aumentare l'usabilità e rendere il tutto più trasparente possibile, ho definito una conversione implicita da `Category` a `ConditionCategoryOps` e ho rinominato il metodo `and` di `ConditionStatement` in `&&`;
- `ConditionStatement` viene arricchito sfruttando il pattern *Pimp My Library*, che abilita alla creazione di una `Rule` specificando un insieme di `Action`.

Il listato 4 riporta un esempio di creazione di una regola sfruttando il dsl.

```
val waterRule = Temperature > 20.2 && Humidity < 30
   execute Water(20 seconds)
```

Listing 4: Esempio d'uso del dsl per la creazione di una `Rule`

Inoltre, ho realizzato la relativa parte di testing per le regole, in particolare `RuleSpec` e `ConditionCategorySpec`.

Successivamente, mi sono occupato dell'implementazione dei concetti principali all'interno del modulo `device`, in particolare di `Sensor`, `Actuator`, i relativi attori e `Operation`. Ho implementato `Sensor` e `Actuator` come delle "rich interface" estendibili in un secondo momento dall'utilizzatore della libreria. Inoltre ho fornito delle *static factory* per consentire la creazione rapida di `Sensor` e `Actuator` senza necessariamente coinvolgere l'ereditarietà. Ad esempio, è possibile creare un `Sensor` a partire da uno `Stream` scala. Ho implementato, poi, `ActuatorActor` e `SensorActor` e il loro comportamento comune in `DeviceActor`. Una migliore implementazione sarebbe stata possibile sfruttando *akka-typed* che porta a livello di dominio il concetto di *behaviour* in stile funzionale. Una importante nota implementativa va ad `Operation` e alla sua gestione; infatti, per evitare problemi di concorrenza legati al completamente differito di un'azione fisica, si è fatto affidamento sul flusso di controllo degli attori, in particolare su `ActuatorActor`. Quest'ultimo, infatti, sfrutta i timer per gestire le operazioni asincrone al fine di cambiare stato ad `Actuator`.

7 Retrospettiva

L'approccio Agile del metodo Scrum unito ad un utilizzo di Git che seguisse l'approccio Git Flow si è rivelato essere estremamente utile nello sviluppo di funzionalità diverse ma spesso interdipendenti, in quanto ha permesso di evitare conflitti pur lavorando in concorrenza. Durante gli sprint settimanali, i membri del gruppo hanno portato avanti lo sviluppo dei task a loro assegnati e si sono incontrati tramite piattaforma Teams per la realizzazione delle parti comuni e per i diversi meetings.

7.1 Sprint 1

Sviluppo Il primo sprint è stato principalmente svolto in presenza e si è incentrato sul setup del progetto e sull'analisi del problema. Durante lo svolgimento sono stati inizializzati i tool inerenti al processo di sviluppo, in particolare:

- Gradle, per la gestione delle dipendenze;
- GradleProject, per la suddivisione dei task;
- TravisCI, per la continuous integration.

È stato inoltre iniziato il design architetturale.

Considerazioni Le difficoltà di questa fase iniziale sono molto legate all'incertezza dei requisiti per un dominio nuovo e non standardizzato. Nel prossimo sprint si prevede di terminare il design architetturale e quello di dettaglio e di iniziare le prime fasi implementative.

7.2 Sprint 2

Sviluppo Questo sprint è il primo che si svolge da remoto e vede il progredire e il concludersi della fase di design.

Considerazioni La fase di design ha preso più tempo del previsto e ha generato malumori nel team che si è trovato ad affrontare problematiche inattese. Il prossimo sprint inizierà con le prime fasi implementative da portare avanti in comune.

7.3 Sprint 3

Sviluppo Questo sprint vede l'inizio della fase implementativa con lo sviluppo delle interfacce core del framework e la configurazione di *Akka Remote*. Sono state inserite le user stories nel Product Backlog che verrà mantenuto su GitHub e dettagliato con il progredire del progetto.

Considerazioni L'inizio dell'implementazione ha riportato energia nel team. Sono state dettagliate le prime user stories individuando i task da implementare nello sprint successivo.

7.4 Sprint 4

Sviluppo Questo sprint inizia con la suddivisione dei task da implementare. Si inizia a realizzare tutto ciò che riguarda la zona e la sua gestione (`ZoneManagerActor`, `ZoneActor` e relativo interfacciamento con il client); si iniziano a concretizzare sensori e attuatori e la loro gestione lato server i.e. `EntityManagerActor`.

Considerazioni La prima settimana di sviluppo non ha subito intoppi o ritardi. Si dettagliano ulteriori user stories per lo sprint successivo.

7.5 Sprint 5

Sviluppo Il focus di questo sprint è legato ai concetti di aggregatore e stato, regole e implementazioni in Prolog.

Considerazioni Nonostante qualche incertezza iniziale sul come suddividere il lavoro in Prolog, lo sprint è progredito in maniera spedita e ha visto il completamento delle principali feature previste. Il prossimo sprint vedrà l'implementazione della demo e gli ultimi refactoring.

7.6 Sprint 6

Sviluppo Questo è l'ultimo sprint e converge alla realizzazione di quanto previsto.

7.7 Considerazioni finali

Il progetto è stato realizzato utilizzando la metodologia Scrum, anche se questa è risultata poco consona per via del fatto che lo scope e i requisiti non fossero ben definiti da subito. Tutti i membri del team hanno utilizzato per la prima volta la metodologia Scrum, che ha necessariamente portato a rivedere le proprie abitudini di sviluppo. Le principali difficoltà riscontrate sono state relative all'organizzazione del lavoro, in particolare nella corretta suddivisione in sprint, soprattutto laddove erano presenti dipendenze. Nonostante un'inerzia iniziale dovuta alla nuova metodologia, il team ha cercato di organizzare al meglio il lavoro riuscendo a concludere il progetto entro i tempi previsti e rispettando i requisiti prefissati. Il progetto, in generale, si è rivelato di grande utilità per la crescita dei membri del team, che hanno potuto migliorare le proprie abilità in Scala e apprezzare la flessibilità che il linguaggio offre. Abbiamo inoltre approfondito la programmazione logica introducendo Prolog in fase di implementazione. L'utilizzo

dei test si è rivelato di grande efficacia e importanza sopratutto in combinazione con la *Continous Integration*. Infatti, durante le fasi di refactoring, i test di regressione hanno permesso di individuare facilmente le rotture del sistema a fronte di modifiche. Abbiamo apprezzato anche il supporto offerto da *Scalatest*, che ha permesso la stesura di test chiari e ben organizzati.

Alcune decisioni architetturali hanno impattato negativamente sul design di dettaglio e sull' implementazione. Ad esempio, una pecca della nostra attuale soluzione è legata all'impossibilità di definire sensori ed attuatori a livello globale. Dai requisiti si evince infatti che ogni dispositivo intelligente può essere associato al massimo ad una zona. Ad esempio, sensori fisici come quello meteo, potrebbero essere considerati come globali ed essere mappati in un unico sensore logico associabile a tutte le zone. Durante la realizzazione della soluzione, essendo il prodotto finale un framework e non un'applicazione, siamo stati spesso portati a far prevalere soluzioni che garantissero la massima usabilità del framework rendendo, però il nostro compito più complesso ed articolato. Per di più, l'intenzione di creare un framework che fosse meno confinato possibile, ci ha reso difficile l'implementazione sotto diversi punti di vista. Si pensi ad esempio che la volontà di non fissare le `Category` e `Action` ha reso il loro design e la loro implementazione un po' "tricky".