HOME: House Optimized Monitored Environment

Manuele Pasini
manuele.pasini@studio.unibo.it

Konrad Gomulka
konrad.gomulka@studio.unibo.it

Luca Salvigni
luca.salvigni7@studio.unibo.it

Andrea Sperandio
andrea.sperandio4@studio.unibo.it

31 agosto 2020

Indice

Capitolo 1

Introduzione

Il progetto HOME nasce con l'idea di poter realizzare un sistema in grado di controllare una serie di dispositivi presenti all'interno di un'ipotetica abitazione tramite un'applicazione desktop che permetta di visualizzarne lo stato, modificarlo o pianificarne la modifica in base al verificarsi di determinate condizioni; a questo scopo è stato redatto il seguente caso di studio.
"All'interno di un progetto privato di investimento in campo edile, si vuole realizzare una nuova zona condominiale all'interno della quale ogni abitazione sarà provvista di un sistema software che permetta ai condomini di associare ad esso una serie di dispositivi IoT mirati ad agevolare e migliorare l'esperienza di vita domestica.
Ogni abitazione sarà dotata di un sistema di riscaldamento/climatizzazione centralizzato e di un termometro atto a rilevare la temperatura esterna; allo stesso modo ogni stanza di un'abitazione sarà dotata di almeno una lampadina e di tre sensori: temperatura, umidità ed un rilevatore di movimento in grado di determinare se la stanza è vuota o meno. L'utente deve essere in grado di monitorare consumi e lo stato dei propri dispositivi in ogni momento.
ll sistema è composto da un coordinatore ed una serie di device IoT che l'utente gli può associare. Una volta avvenuta l'associazione, l'utente deve poter visualizzare in tempo reale ed in qualunque momento i consumi, lo stato e gli aspetti programmabili del componente: in particolare lo spegnimento, l'accensione e le eventuali funzionalità specifiche.
Allo stesso modo il sistema deve prevedere l'esistenza di "profili di utilizzo": dicasi profilo una serie di regole e/o impostazioni dei device presenti all'interno del sistema che portano ad un comportamento ben definito dello stesso; si prenda il caso del profilo "Estate": tale profilo fa sì che ogni lampada collegata rimanga spenta durante le ore diurne, che il condizionatore venga acceso durante il giorno se la casa non è vuota, o, in caso contrario, tre ore la sera. Esistono due tipologie di profili:

- preimpostati e già presenti nel sistema;
- programmabili ad hoc dall'utente."

Capitolo 2

Processo di sviluppo

2.1 SCRUM

Per lo sviluppo di questo progetto si è scelto SCRUM come metodologia di organizzazione del lavoro, seguendo tale metodologia sono riportati sotto i ruoli assegnati ai singoli componenti del gruppo:

- **SCRUM Master** : Konrad Gomulka
- **Product Owner**: Manuele Pasini
- **Development team**: Konrad Gomulka, Manuele Pasini, Luca Salvigni, Andrea Sperandio.

Nel primo meeting tenutosi tra i componenti del team si è stilato un caso di studio dal quale è stato possibile estrapolare i vari requisiti di progetto ed individuare le componenti fondamentali sulle quali il sistema si basa. A questo punto è stato effettuato un primo sprint planning della durata di una settimana: si è cercato di massimizzare le ore di interazione e brain storming tra i componenti del gruppo in modo da aver prodotto, al termine della sprint, il product backlog e l'architettura del sistema, oltre ad un mockup da esporre al committente.
Svolta questa prima fase, tutti gli sprint seguenti sono stati portati avanti seguendo la metodologia *SCRUM* vista a lezione. La revisione in itinere dei task è stata effettuata tramite discussioni orali per mezzo della piattaforma *Discord*; al termine di ogni task la discussione è stata iterata per verificarne l'effettivo completamento e per la suddivisione dei compiti successivi. Si è cercato inoltre, durante le sprint, di mantenere costante e frequente l'interazione tra i componenti del gruppo in modo da limitare il più possibile eventuali ambiguità o problematiche di comunicazione tra i membri, cercando di procedere sempre come un'unica entità allo stesso passo.

2.2 Travis CI

Travis CI è una piattaforma di integrazione continua, grazie alla quale si riesce a supportare il processo di sviluppo di un progetto creando e testando automaticamente le modifiche al codice, fornendo poi un feedback sul successo del cambiamento. Quando si esegue una build, Travis clona il repository GitHub che gli è stato inizialmente collegato, in un ambiente virtuale, ed esegue una serie di attività per testare il codice. Alla conclusione della build, i feedback restituiti possono essere: passed, errored, failed, canceled. Questa piattaforma è risultata davvero molto utile, specificando in che zona di codice si verificassero errori in caso di fallimento della build, aiutando di conseguenza il team ad intervenire per la correzione di eventuali bug o errori.
LINK a Travis CI.

2.3 Trello

Trello è un software gestionale gratuito utilizzato per il project management. Possiede tre elementi chiave che sono:

- Board: una semplice lavagna che rappresenta l'organizzazione del progetto.
- List: ce ne possono essere diverse all'interno della board, e rappresentano i macro-step effettuati all'interno di un progetto (es. "Da fare", "In lavorazione, "Fatto").
- Card: si tratta di una singola unità di base della board che rappresenta un singolo task da compiere o un'idea.

Trello è risultato davvero utile per organizzare le sprint settimanali, aggiungere eventuali idee future e a suddividere i task da portare a termine entro fine sprint per ogni membro del team. Quando un'attività era completata, quest'ultima veniva aggiunta alla sezione "Fatto". Alla fine di ogni sprint, dopo aver stabilito quella successiva, si aggiornava Trello in modo da rendere chiaro ciò che ogni membro doveva svolgere durante la settimana. Nel caso in cui un task non fosse stato terminato entro la settimana prestabilita, magari a causa di qualche errore di valutazione, quest'ultimo veniva aggiunto alla sezione "In esecuzione".
LINK alla board Trello.

2.4 GitHub

GitHub è una piattaforma di hosting web open source che permette di ospitare progetti software tramite un software di controllo di versione chiamato Git. Quest'ultimo infatti è stato reso necessario, oltre che comodo e affidabile, siccome permette di gestire e controllare gli aggiornamenti di un progetto senza sovrascrivere alcuna parte del progetto stesso. GitHub è risultato davvero utile garantendo la possibilità di avere tutte le vecchie versioni conservate nel repository, così da poter recuperarle in caso di necessità, come per esempio

bug o errori.
LINK alla repo GitHub.

2.5 GitFlow

Come flusso di sviluppo tramite Git è stato deciso dal team di usare i rispettivi branch:

- Master: utilizzato unicamente per le release
- Develop: utilizzato per lo sviluppo delle funzionalità principali, ogni altro branch una volta ritenuto finito effettua merge con Develop
- Gui: branch creato per sviluppare la GUI
- feature-CustomProfileSensorUpdate: utilizzato per sviluppare una funzionalità dei profili custom
- travis-mosquitto-test: utilizzato per testare le varie configurazioni di docker, docker compose, travis e mosquitto in modo da produrre dei files di configurazione in grado di automatizzare il processo di build, test e deploy. E' stato utilizzato soprattutto per risolvere le difficoltà dovute al test e deploy di un'applicazione desktop con interfaccia grafica distribuita tramite immagine docker.

2.6 Docker

Docker è stato utilizzato per automatizzare il deployment dell'applicazione tramite immagine. E' stato usato Travis CI per automatizzare, oltre alle fasi di build e di test, la creazione dell'immagine dell'applicazione e la sua pubblicazione su un Docker Hub accessibile dall'esterno. Questa è poi usata da Docker Compose per lanciare l'applicazione, che dipende appunto dall'immagine e dal servizio broker mosquitto.
LINK al Docker Hub.

2.7 Tests

Ogni fase di scrittura del codice è stata seguita da una di testing, ossia da una fase di scrittura di tests volti a valutarne la correttezza e la capacità di integrazione con codice preesistente. Questo ha permesso di scoprire bugs ed incongruenze sin da subito, oltre che di pensare più dettagliatamente al funzionamento previsto del codice scritto. Particolarmente utile si è rilevata durante le fasi di integrazione branches e refactoring, fornendo un valido strumento di controllo.

La code coverage dei tests forniti indica che i tests coprono il 40% delle classi, il 60% dei metodi ed il 47% delle linee di codice totali.

Capitolo 3

Requisiti

3.1 Requisiti di business

Le funzionalità che il sistema HOME deve tassativamente esporre:

- possibilità di monitorare la propria abitazione tramite dispositivi connessi in rete;
- gestione autonoma e libera dei dispositivi da parte dell'utente (aggiunta/rimozione/modifica);
- monitoring in tempo reale dello stato dei dispositivi;
- possibilità di impostare un comportamento definito (profilo) al sistema.

3.2 Requisiti utente

Identificano le operazioni che l'utente potrà compiere sul sistema, sotto descritte come user stories:

- come condomino, voglio poter gestire autonomamente le stanze presenti nella mia abitazione, per questo motivo necessito di poter aggiungere o rimuovere stanze al sistema;
- come condomino, voglio poter gestire autonomamente i dispositivi presenti all'interno delle mie stanze, per questo motivo necessito di poter aggiungere o rimuovere dispositivi e di sfruttarne appieno le funzionalità;
- come condomino, voglio poter monitorare lo stato della casa, per questo devo essere in grado di osservare lo stato dei dispositivi in tempo reale;
- come condomino, voglio poter creare dei profili personalizzati in modo da poter scegliere un comportamento che il sistema deve tenere secondo un mio insieme di regole;
- come condomino, voglio poter accedere e modificare lo stato di ogni singolo dispositivo.

3.3 Requisiti funzionali

Identificano più dettagliatamente le operazioni che il sistema deve essere in grado di eseguire:

- lista di funzioni ad accesso rapido (profili) non personalizzabili: tramite l'interfaccia grafica, l'utente dovrà poter essere in grado di impostare un profilo attivo tra quelli presenti nel sistema; un profilo può essere attivato solo e solo se tutti i dispositivi menzionati da tale profilo sono presenti all'interno dell'abitazione;
- creazione di profili: tramite interfaccia grafica, oltre a scegliere tra quelli esistenti all'interno del sistema, l'utente deve potere creare profili ad-hoc. Tali profili possono definire il comportamento di tutti i dispositivi connessi al sistema;
- modifica dello stato dei dispositivi connessi tramite profili o input utente (attraverso interfaccia grafica);
- gestione clima: la climatizzazione ed il riscaldamento sono gestiti internamente stanza per stanza, così come la temperatura e l'umidità interna;
- gestione luminosità in base a condizioni atmosferiche interne/esterne (gestisce luci e serrande): in base alle condizioni meteo, il controllore dovrà impostare l'intensità di certe lampadine presenti all'interno della casa;
- Statistiche sui consumi: il sistema deve offrire all'utente la possibilità di monitorare in tempo reale i consumi dei dispositivi connessi;
- riflessione delle modifiche: l'interfaccia grafica deve essere in grado di mostrare solo gli aggiornamenti completati con successo, eventuali aggiornamenti falliti non devono essere mostrati.

3.4 Requisiti non funzionali

Rappresentano gli aspetti qualitativi che il sistema deve rispettare:

- login: dovendo controllare dati ed informazioni sensibili, è necessario garantire all'utente una misura di sicurezza che escluda accessi non desiderati tramite una funzione di login;
- sicurezza connessione dispositivi: allo stesso modo degli accessi, è necessario prevenire l'eventuale tentativo di associazione di device maligni o non autorizzati all'interno del sistema; a tale scopo è prevista una password che ogni dispositivo deve inserire al momento della connessione al sistema; tale password deve essere robusta e tenuta segreta.
- controllo dispositivi: lo stato di un dispositivo può essere modificato solamente in tre modalità ben definite:

  - tramite un profilo;
  - tramite input utente;
  - dal sistema stesso a frutto di una rilevazione da parte di un sensore.

- robustezza: essendo una rete locale il mezzo di comunicazione tra i componenti del sistema, è necessario prevedere meccanismi di fault-tolerance: nel nostro caso, escludendo gli aspetti di sicurezza descritti sopra, la minaccia maggiore alla stabilità del sistema è la disconnessione/perdita di connessione di un dispositivo o del controllore.

3.5 Requisiti implementativi

- Un primo requisito implementativo è dato dal dovere identificare l'insieme dei dispositivi associabili al sistema:
  - lampadina;
  - condizionatore/deumidificatore;
  - serrande (porte, garage, finestre);
  - caldaia;
  - TV;
  - lavatrice
  - asciugatrice;
  - lavastoviglie;
  - forno;
  - impianto stereo.

- Una volta costruita l'insieme dei dispositivi è necessario stabilire quali operazioni gli utenti potranno effettuare su essi, ovvero le funzionalità messe a disposizione da ogni dispositivo. Si presume che ogni dispositivo abbia funzionalità di accensione/-spegnimento e una statistica sul consumo:

Device	**Funzionalità**
Lampadina	Intensità
Condizionatore/deumidificatore	Aumenta/diminuisci/imposta temperatura
Serrande	Aperto/chiuso
Caldaia	Imposta temperatura
TV	Volume
Lavatrice/asciugatrice/lavastoviglie	Piani predefiniti di lavoro
Forno	Modalità funzionamento, temperatura
Impianto stereo	Volume

- Per la realizzazione dell'interfaccia grafica si è scelto di studiare ed utilizzare scala-swing[1], una libreria sviluppata dal team di sviluppo di Scala che permette un approccio ed un utilizzo più funzionale alla libreria Java-swing [1]: questa scelta è prettamente accademica e sperimentale in quanto si è preferito l'idea di provare ad applicare il paradigma funzionale anche alla parte grafica piuttosto che rivedere aspetti già visti e non inerenti al corso;
- Si è reso necessario poi, dovendo i vari dispositivi essere in grado di reagire ad eventi inerenti ad alcune condizioni atmosferiche (temperatura, umidità, etc), costruire un modello in cui tali condizioni atmosferiche siano simulate e modificabili tramite applicazione;
- è necessaria la costruzione di un'infrastruttura di rete in grado di permettere ai dispositivi di comunicare con il sistema e viceversa, rispettando la qualità di *robustezza* definita a 3.4

[1] `https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html`

Capitolo 4

Design architetturale

4.1 Componenti

Al seguito di uno studio effettuato durante la prima sprint, il sistema è stato scomposto e sono state individuate tre componenti core:

- dispositivi (device);
- coordinatore;
- interfaccia grafica (GUI).

Un device rappresenta un dispositivo simil-IoT sul quale l'utente può effettuare operazioni di modifica o di pianificazione; concettualmente è rappresentato da un'entità a sé stante separata dal resto del sistema (e che dunque necessita di connettersi ad esso) e **reattiva**: deve essere in grado di reagire agli eventi che si verificano all'interno dell'abitazione modificando il proprio stato.
I device sono a loro volta stati scomposti in due sottocategorie: la prima racchiude tutti i dispositivi elencati a 3.5 e dotati della reattività descritta sopra; la seconda categoria racchiude i **sensori**: questi sono device a tutti gli effetti ma non godono di reattività, non sono in grado di conseguenza di reagire agli stimoli dell'ambiente circostante; si è deciso di utilizzare la quaterna di sensori

- fotometro: rileva intensità luminosa;
- termometro: rileva temperatura;
- igrometro: rileva umidità;
- sensore di presenza: rileva presenza o meno all'interno di una stanza.

Per quanto sia netta la differenza tra queste due categorie di device, il loro lavoro è complementare: i sensori si occupano di simulare le condizioni atmosferiche interne e di conseguenza generare gli stimoli ai quali i device sono in grado di reagire. La definizione di "simil-IoT" deriva dal fatto che pur non avendo in dotazione e di conseguenza non aver

utilizzato veri e propri dispositivi IoT in commercio, quelli realizzati ed integrati nel nostro sistema rimangono comunque dispositivi connessi dotati di logica, attuatori (rappresentati dalle funzionalità offerte da ogni singolo device) e una serie di sensori (elencati qui sopra). Le restanti due componenti core sono strettamente legate tra loro, rappresentano la parte di sistema esposta all'utente e con la quale l'utente interagisce: più specificatamente l'utente interagisce con la GUI, la quale si occupa di notificare il coordinatore che si impegna a comunicare con il device fisico. E' fondamentale che l'interfaccia grafica sia aggiornabile in ogni momento e rifletta in tempo reale lo stato dei dispositivi connessi mostrandone le proprietà. Tramite interfaccia grafica, oltre a poter verificare in ogni momento lo stato dei dispositivi, l'utente deve avere anche la possibilità di vedere tutte le stanze della casa, riuscendo ad interagire quindi con un device di una stanza specifica, garantendo di fatto una migliore usabilità. Sempre attraverso GUI, si ha la possibilità di creare dei profili, ovvero una serie di comportamenti logici tramite i quali l'utente può applicare un insieme di regole, che verranno applicate o all'attivazione del profilo stesso o quando verranno soddisfatti dei determinati criteri. Per esempio nel caso in cui ci siano meno di 15°C in salotto, si vuole accendere l'aria condizionata in salotto ad una temperatura di 25°C. Questo risulta una delle parti chiave del progetto, in cui si può applicare l'automaticità del sistema, garantendo la possibilità che gli stati dei device vengano modificati secondo certe regole impostate dall'utente, senza che quest'ultimo interagisca direttamente con l'interfaccia grafica stessa.

Una volta definite le componenti core, un altro punto indirizzato nella prima fase di studio è stato quello di identificare e standardizzare il flusso di controllo all'interno del sistema: lo stato iniziale del sistema deve mostrare tutti i dispositivi connessi e il loro relativo stato; dallo stato iniziale deve essere possibile modificare lo stato dei dispositivi in due modalità:

- manuale: lo stato dei dispositivi viene modificato manualmente tramite interfaccia grafica dall'utente;
- automatica: lo stato dei dispositivi viene modificato automaticamente dal coordinatore attraverso i profili.

Il ruolo principale del Coordinatore è appunto di gestire la modalità automatica. Esso definisce tutta la logica mediante la quale vengono applicati i profili, sia che essi siano preimpostati, e quindi definiti dai programmatori, che custom, ovvero creati a runtime dall'utente. Il Coordinatore fornisce anche dei metodi mediante i quali la GUI può interagire con i device nonché creare dei profili basandosi sulle scelte dell'utente. Ultima funzione offerta dal coordinatore è il Logging dei dati trasmessi dai device.

4.2 Interazione

Per quanto riguarda l'interazione tra dispositivi e coordinatore, si è deciso di sviluppare ed integrare un'architettura a scambio di messaggi; nel dettaglio la scelta è ricaduta su MQTT (Message Queue Telemetry Transport)[2].

MQTT è un protocollo a scambio di messaggi che implementa un'architettura di tipo "publish-subscribe": contrariamente ad un'architettura nella quale i messaggi vengono inviati ad uno o più destinatari specifici, all'interno di un'architettura "publish-subscribe" i messaggi vengono organizzati in classi dette *topic*, la cui struttura può essere riconducibile a quella di un albero dalla cui radice si ramificano delle gerarchie di topic sempre più specifiche:

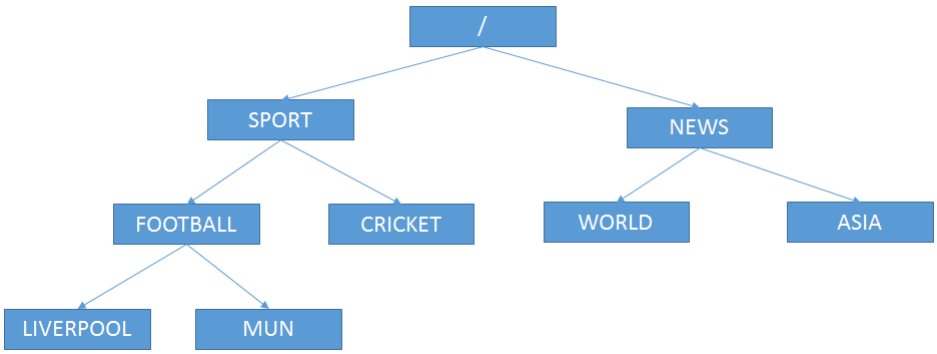

Figura 4.1: Esempio di gerarchia di topic

Le entità in gioco in MQTT sono due: i *client* (nel nostro caso i dispositivi e il coordinatore) e un *broker*. I client si sottoscrivono ai topic per i quali vogliono ricevere aggiornamenti. Una volta registrati, i client possono pubblicare o ricevere messaggi inerenti ai topic ai quali si sono iscritti. Il broker è un componente software che si occupa di ricevere e smistare i messaggi. In particolare, quando un client invia un messaggio su una data topic, il broker lo raccoglie e lo inoltra a tutti i clients sottoscritti a quella topic.
Una prima motivazione per la quale si è optato per questa scelta è dovuta dalla possibilità di avere un numero elevato di dispositivi connessi: invece di dover gestire la comunicazione diretta tra dispositivo e coordinatore è preferibile aprire semplicemente delle connessioni verso il broker. Infatti in questo modo la gestione delle comunicazioni e degli aspetti relativi alla rete è affidata ad un modulo software dedicato, che permette anche trasmissioni broadcast. Vengono sotto elencate ulteriori motivazioni che hanno portato a questa scelta:

- è un protocollo asincrono nato per risolvere problemi in sistemi real-time ed embedded, in cui la connessione è inaffidabile;
- utilizza TCP/IP (Transmission Control Protocol) come protocollo di livello di trasporto e può adottare connessioni di tipo TLS/SSL (Transport Layer Security/Secure Sockets Layer);
- è supportato da diverse librerie, disponibili anche per Scala;
- fornisce un'interazione client-server-client, in cui ogni dispositivo è client e comunica solo con il Broker centralizzato;
- è un protocollo leggero (più di HTTP) e semplice da utilizzare.

Capitolo 5

Design di dettaglio

5.1 Device

I device sono stati creati a partire da un'interfaccia base chiamata Device contenente i tratti comuni a tutti i device, ignorando se essi siano simulati o meno.
Ogni device possiede un proprio DeviceType ovvero un trait definito da vari case object (simile ad un enum) il quale offre varie funzioni che permettono ad esempio di stabilire se si tratta di un sensore o attuatore, sfruttando anche il pattern pimp my library per effettuare i confronti o cercare elementi nei Traversable.

Per simulare i Device non presenti in locale ma interconnessi mediante rete con il protocollo MQTT è presente una classe AssociableDevice contenente tutte le variabili ed i metodi utili al protocollo. Ciascuno di questi dispositivi è in grado di gestire i messaggi basilari in arrivo dal Coordinatore nonché quelli specifici per la propria tipologia di dispositivo mediante il metodo handleDeviceSpecificMessage.

Varie tipologie di dispositivi hanno dimostrato di possedere delle caratteristiche in comune pertanto la classe finale di un Device è realizzata come mixin tra Device, AssociableDevice e i relativi trait che forniscono funzioni utili alla tipologia di Device considerata, un esempio può essere il trait ChangeableValue, il quale offre la possibilità di implementare un valore numerico variabile e con un valore minimo e massimo possibile.

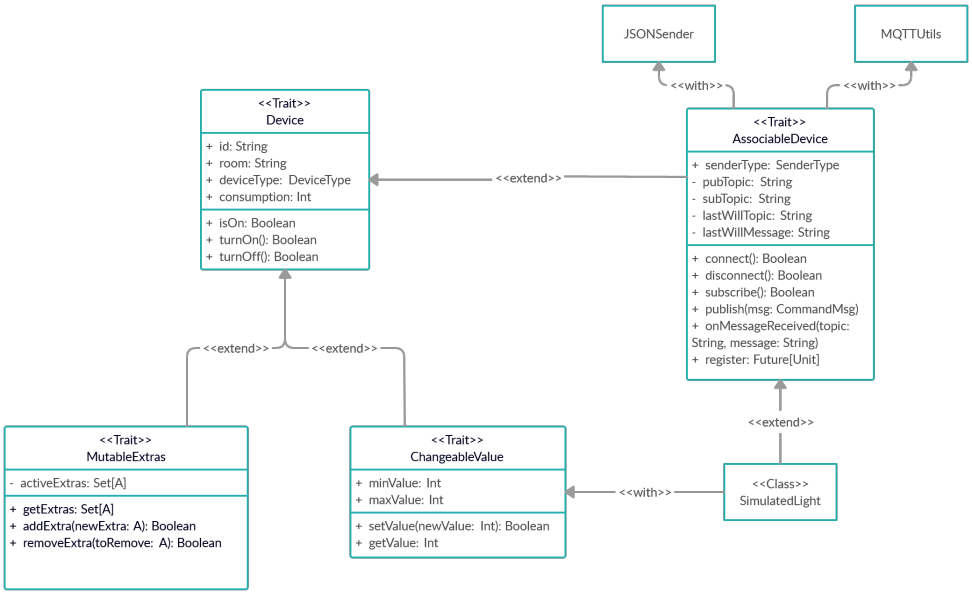

Figura 5.1: Struttura Device

Tale trait è implementato da un dispositivo Lampadina, ma anche da uno Stereo o un Climatizzatore.
L'istanziamento di un Device è possibile utilizzando il pattern Factory all'interno dell'apply dell'oggetto Device o richiamando direttamente il dispositivo della tipologia desiderata tramite il relativo metodo apply.

5.2 Coordinatore

Il Coordinatore, come suggerisce il nome, si occupa di coordinare la comunicazione con i vari Device e di gestire i messaggi in arrivo. Esso è un oggetto Singleton contenente un Set di tutti i device connessi al sistema con i relativi metodi per gestirli, le istruzioni in base alle quali reagire una volta ricevuto un messaggio, dei metodi per gestire il profilo attivo e delle funzioni per ricevere i dati statistici sul funzionamento del sistema.

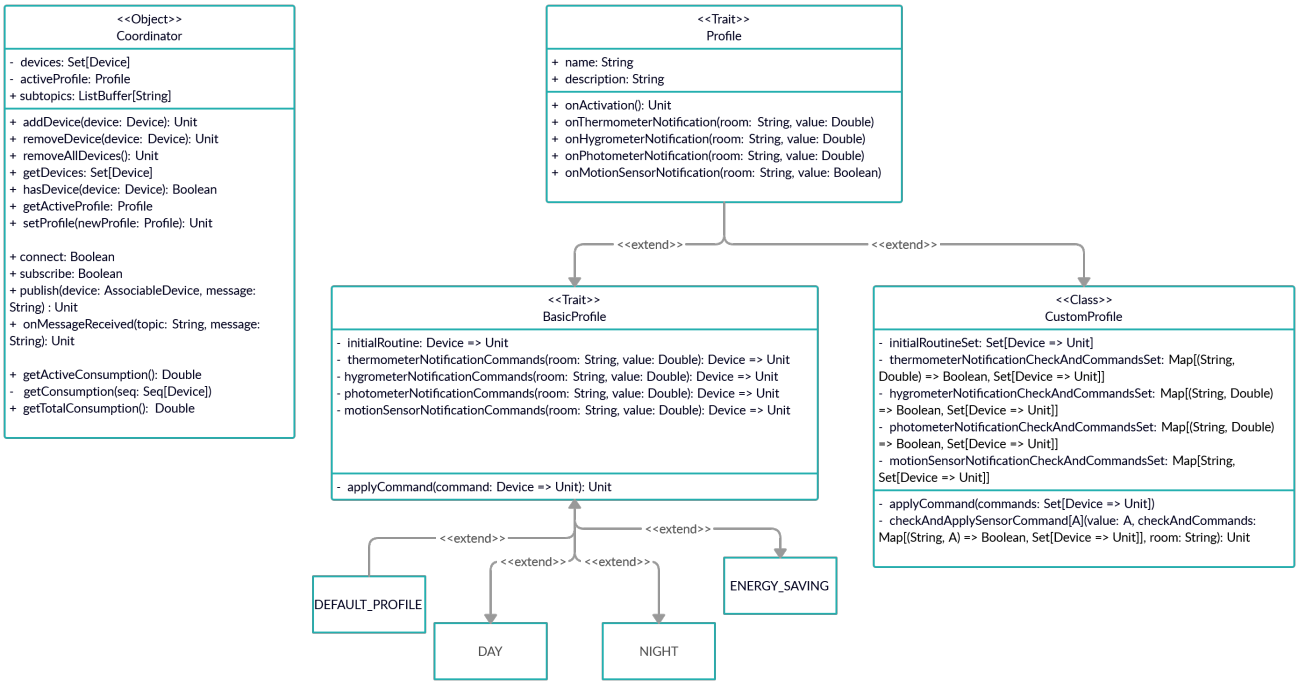

Figura 5.2: Struttura Coordinatore e Profile

5.2.1 Profili

Per quanto riguarda i profili essi sono composti da funzioni che verranno applicate ai device, ognuna di queste funzioni è un match case che se rispettato modifica i vari device secondo le regole definite dal profilo attivo. Essi si dividono in due categorie:

- Profili Preimpostati: Sono i profili definiti dallo sviluppatore del sistema, di base sono [DEFAULT PROFILE, DAY, NIGHT, ENERGY SAVING].
- Profili Custom: Sono i profili creati dall'utente che può scegliere le istruzioni da applicare una volta caricato il profilo nonché in reazione ai valori ricevuti dai sensori.

5.2.2 Log e statistiche

In quanto a statistiche il Coordinatore offre due funzioni in grado di calcolare il consumo attuale nonché il consumo totale dell'abitazione dal momento della sua accensione. Per memorizzare i vari dati di log è stato utilizzato un semplice CSV poiché ritenuto adatto per la scala del sistema e per motivi di performance. Ciononostante un possibile sviluppo futuro può implicare l'utilizzo di un database e/o l'implementazione di Apache Spark per la gestione dei dati statistici.

5.3 GUI

L'interfaccia grafica è la parte di sistema che si occupa di mostrare lo stato dei dispositivi connessi e permette agli utenti di interagirci. Il primo passo nella definizione di GUI è

stato quello dello studiarne il ruolo all'interno del sistema e di conseguenza modellarne la struttura che questa deve rispettare: la continua interazione tra quest'ultima e il coordinatore ha fatto si che si optasse per una realizzazione di un'istanza singola di GUI, istanziata una volta sola ad inizio progetto e richiamabile in qualunque punto.

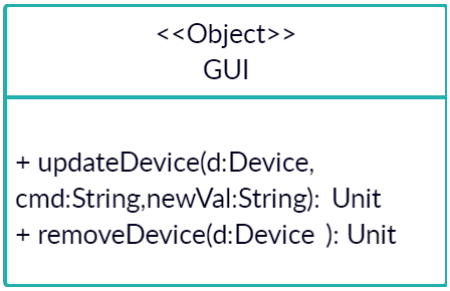

Figura 5.3: Struttura principale GUI

5.3.1 Modellazione device

Modellata la struttura base, ci si è occupati del contenuto, indirizzando subito la problematica della rappresentazione dei dispositivi: data la natura simulativa di questo software, all'interno del quale viene fatto il deploy dei device e l'instanziamento GUI, è evidente che quest'ultima sarebbe in grado di ottenere i riferimenti ai device connessi e potrebbe aggiornare l'interfaccia grafica accedendo direttamente ad essi; tuttavia questa scelta romperebbe l'architettura definita a 4.2 secondo la quale è il coordinatore a comunicare con i dispositivi. Fatte queste considerazioni, si è optato per mantenere una **copia** di tutti i device esistenti e connessi al sistema; questa decisione permette a GUI di riuscire a rappresentare i device basandosi su un'entità complessa e non tramite semplici valori da rappresentare. Questa scelta comporta inevitabilmente che ogni qual volta un device connesso modifichi il proprio stato, tale modifica deve essere riflessa anche sui dispositivi copia mantenuti all'interno di GUI. questa decisione è stata facilitata dal design di Device (5.1), che permette sia di avere dispositivi connessi (AssociableDevice) che dispositivi non connessi (Device), che sono quelli contenuti in GUI.

5.3.2 Room

Al contrario di quanto definito all'interno del coordinatore, dove la *stanza* all'interno della quale è collocato un device è una mera proprietà del device stesso che non viene utilizzata se non per operazioni di filtraggio e dunque non necessita di un'ulteriore modellazione, all'interno di GUI la *stanza* assume un ruolo più importante e diventa un concetto da modellare; questo perché GUI è il punto d'incontro tra l'utente e il sistema software e deve quindi essere in grado di rappresentare ciò che il sistema computa e allo stesso tempo presentarlo all'utente nella maniera più semplice possibile garantendo la migliore user experience. La soluzione è stata individuata nella modellazione di due entità

- Room: trait[1] che incapsula una serie di comportamenti che una stanza deve rispettare;
- GUIRoom: rappresentazione grafica di una stanza

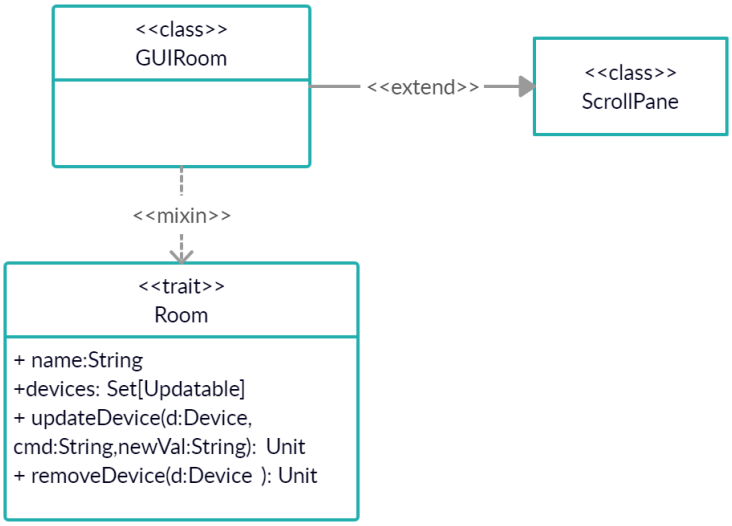

Figura 5.4: Struttura GUIRoom

Come mostrato in figura 5.4, una GUIRoom contiene al suo interno un'insieme di device identificati dal trait "Updatable"; è necessario spendere qualche parola a giustificare tale scelta.
Sono state individuate precedentemente (vedi ultime righe 4.1) due modalità di funzionamento del sistema: *manuale* e *automatica*. Questa distinzione è riflessa anche nel comportamento di GUI, all'interno della quale esistono due procedure per modificare un device, implementate tramite il trait "Updatable" e tramite il trait "EditableFeature".
Il trait "Updatable"

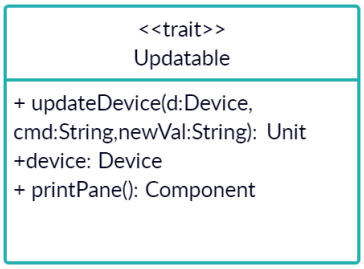

Figura 5.5: Trait rappresentante il comportamento che un l'implementazione grafica di un device deve rispettare

rappresenta le proprietà che un qualunque dispositivo grafico deve implementare per potere ricevere modifiche dal coordinatore ed essere visualizzato all'interno di GUI.

[1] `https://docs.scala-lang.org/tour/traits.html`

Dicasi **proprietà** di un device, un qualunque suo aspetto programmabile dall'utente (es. intensità luminosa per una lampadina); il trait "EditableFeature"

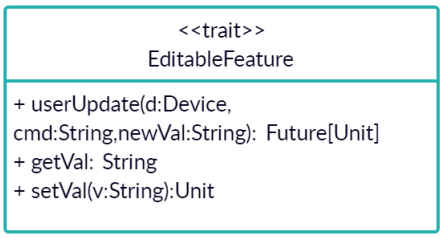

Figura 5.6: Trait rappresentante il comportamento che una proprietà di un device modificabile dall'utente deve implementare per essere in grado di reagire al flusso di modifica generato da un utente

rappresenta il comportamento che una proprietà di un device deve implementare per poter essere rappresentata in GUI. A livello di GUI, sarebbe stato possibile unificare i due flussi in uno singolo, collassando "EditableFeature" all'interno di "Updatable", tuttavia si è ritenuto preferibile mantenere separati questi due aspetti in quanto se è vero che possono esistere dispositivi che non possono essere modificati da parte dell'utente(vedi ad esempio il caso dei sensori la cui unica funzionalità è generare dati), è altrettanto vero che deve essere possibile, per ogni device contenuto in GUI, modificarne lo stato riflettendo un eventuale cambiamento attuato dalla sua controparte connessa e dunque proveniente dal coordinatore. Per questo è necessario che ogni device contenuto all'interno di una stanza implementi il trait *Updatable*.

5.3.3 GUIDevice

Un *GUIDevice* è la rappresentazione grafica più basilare possibile di un device che questo sistema è in grado di mostrare: dotato solamente delle funzionalità di base presenti in tutti i device, lascia che siano le sue implementazioni specifiche per ogni tipo di device a definirne le eventuali proprietà(da qui in poi verranno chiamate *feature* tutte le proprietà modificabili dall'utente tramite GUI). A questo scopo GUIDevice offre un'interfaccia in grado di aggiungere un numero indefinito di feature ad un device, con la restrizione che ognuna di queste implementi il trait *EditableFeature*, in quanto ogni feature di un device modificabile dall'utente deve poter rifletter tale modifica al coordinatore e di conseguenza ai device connessi.

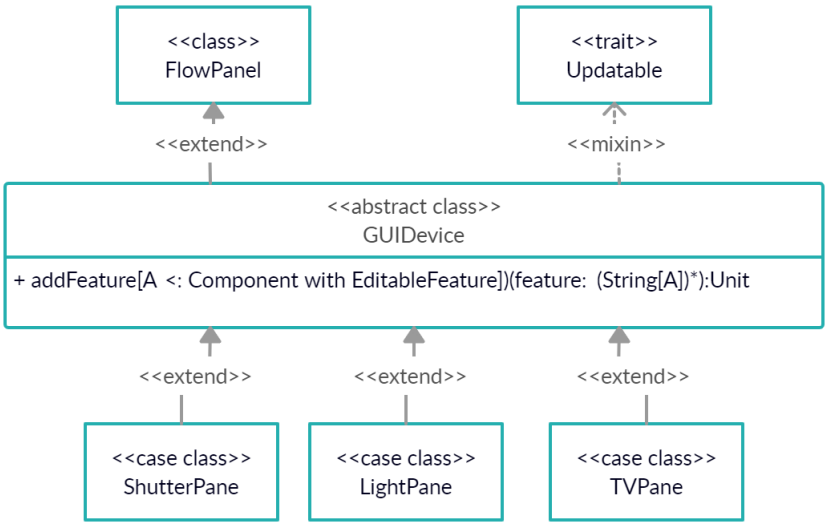

Figura 5.7: Rappresentazione grafica di un device(per motivi di spazio, non sono stati elencati tutti i pane che estendono GUIDevice)

Sono state definite due tipi di feature sufficienti a coprire la totalità delle proprietà esposte dai device:

- BinaryFeature: feature binaria, supporta solamente due valori possibili (es. on/off);
- DeviceFeature: feature non binaria, il range di valori ammessi è maggiore di due e necessità di una rappresentazione più complessa: richiede un componente (che sia in grado ovviamente di rappresentare il range valori ammissibili) che implementi due metodi utilizzati per tenere aggiornato il valore della feature stessa.

La scelta di mantenere separati i concetti di *EditableFeature* e *Updatable* ha portato vantaggi anche in termini di modularità e scalabilità: grazie a questa distinzione è stato possibile costruire la GUI separando completamente a livello logico la struttura dei dispositivi dal loro comportamento atteso (identificato dai trait) e vincolando di conseguenza solamente il comportamento che un device deve implementare, lasciando completamente libera la sua forma.

5.3.4 Sensori e simulazione

Nella sezione 4.1 si è introdotto brevemente il concetto di sensori come simulatori d'ambiente: mentre l'implementazione vera e propria di questo aspetto verrà discussa in 6.3.1, è bene spendere qualche parola su questa decisione.
E' necessario, per potere valutare il comportamento del sistema, che vi sia una parte che permetta di simulare le condizioni e gli stimoli a cui i device reagiscono e modificano il loro stato. Tralasciando la possibilità che un device modifichi il proprio stato sulla base del cambiamento di stato di un altro device (comportamento impostabile tramite un profilo), in linea generale, i dispositivi possono reagire ai cambiamenti di condizioni atmosferiche,

questo significa dover poter comunicare ai dispositivi che sono cambiate ad esempio le condizioni atmosferiche interne, come temperatura ed umidità.
In un sistema applicabile al mondo reale, i sensori sono entità *attive* che ogni Δt inviano un aggiornamento sul valore rilevato e tale valore non può essere impostato da un entità esterna; all'interno del nostro sistema sarebbe stato possibile implementare una soluzione simile, tuttavia si è optato per una soluzione un po' diversa che richiede due ipotesi di partenza:

- GUI non può comunicare con i dispositivi connessi;
- il coordinatore non può effettuare operazioni di modifica sullo stato dei sensori;

Trattandosi pur sempre di una simulazione, la decisione presa è stata quella di affidare a GUI non le copie dei sensori bensì un riferimento diretto ad essi, in questo modo ogni modifica del valore di un sensore tramite interfaccia grafica viene immediatamente riflesso sul sensore connesso, senza passare dal coordinatore. Rispetto all'alternativa proposta poco sopra, questa soluzione lascia all'utente il controllo della simulazione, permettendogli di modificare i valori ambientali a piacere e allo stesso tempo evita di aggiungere una funzionalità di cui il coordinatore per sua natura non è dotato. Una nota negativa derivante da questa scelta è che necessita che i sensori siano direttamente accessibili da GUI, in un eventuale futura versione dove i sensori e i device possono essere istanziati da una macchina diversa da quella in cui viene istanziata GUI, questa architettura non sarà più funzionante e necessiterà di un aggiornamento; nonostante ciò, in questa versione si è preferito dare priorità all'utente finale.

5.4 MQTT

Il protocollo adottato per la comunicazione tra i dispositivi ed il coordinatore è MQTT (Message Queue Telemetry Transport): un sistema di scambio di messaggi di tipo publish-subscribe basato su un modello con Broker.

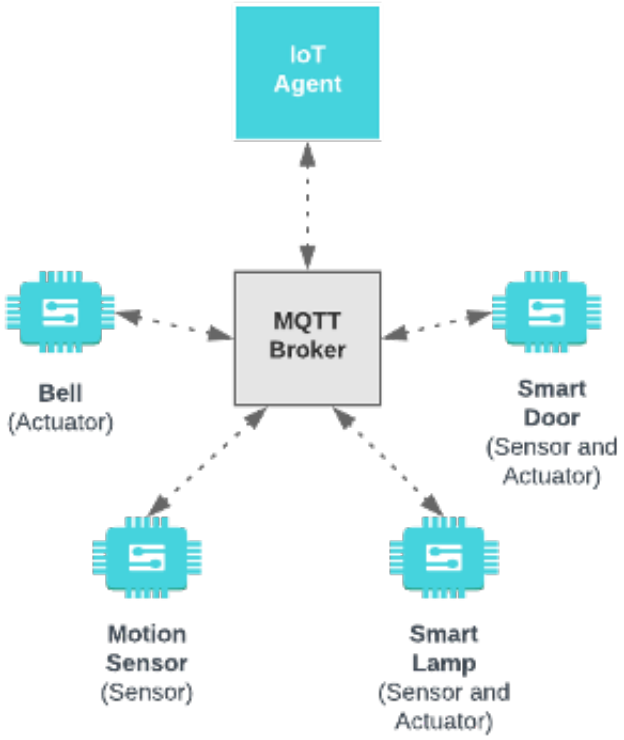

Figura 5.8: Architettura MQTT

Come mostrato dalla Figura 5.8, l'architettura MQTT prevede un broker centrale, con il compito di ricevere e smistare i messaggi, e diversi client, produttori e consumatori dei messaggi. In questo progetto i client MQTT sono rappresentati dai *device*, provvisti di sensori e attuatori, e dal *coordinatore*, mentre come broker MQTT è stato adottato Mosquitto [3].

I sensori quindi pubblicano informazioni relative all'ambiente (publish), quelli con attuatori restano in attesa di comandi da eseguire (subscribe) mentre il coordinatore raccoglie tutti i messaggi, li analizza determinando il comportamento che il sistema deve attuare in risposta alle condizioni ambientali e invia ordini ai device. I primi passi per prendere confidenza con MQTT sono stati mossi in locale tramite un'installazione del broker Mosquitto e l'utilizzo di alcune shells di comandi per simulare i client in comunicazione [4].

MQTT mette a disposizione anche diverse features utili per gestire al meglio la comunicazione tra clients, tra cui:

- livelli di Quality of Service (QoS): garantiscono il livello di sicurezza di completamento in fase di trasmissione/ricezione di un messaggio;
- sessione persistente: permette di evitare l'overhead di una nuova sessione ad ogni riconnessione;
- messaggi retained: fanno sì che un messaggio inviato su un topic sia ricevuto anche da chi successivamente farà subscribe per quella topic;

- last will message: permette di inviare un messaggio automatico su un topic al momento della disconnessione del client.

Molte di queste sono state adottate per la realizzazione del progetto, dopo averne studiato le specifiche di funzionamento [5].

5.5 Formato Dati

I messaggi scambiati tra Coordinatore e Devices sono in formato JSON. Se una entità del sistema riceve un messaggio su una topic sconosciuta o in un formato non valido (non JSON o un messaggio sconosciuto), questo viene scartato.

Il formato JSON scelto rispetta la seguente sintassi:

- 1.1 Device si registra

```
{
  "msg" : "register",
  "sender" : {
    "type" : "device",
    "id" : "A",
    "room" : "Living room",
    "deviceType" : {
      "name" : "LightType"
    },
    "consumption" : 5
  }
}
```

- 1.2 Conferma del Coordinatore

```
{
  "msg" : "regSuccess",
  "sender" : {
    "type" : "coordinator",
    "name" : "Coordinator"
  }
}
```

- 2.1 Il Coordinatore ordina l'accensione del Device

```
{
  "msg" : "0_on",
  "sender" : {
    "type" : "coordinator",
```

```
    "name" : "Coordinator"
  }
}
```

- 2.2 Conferma del Device

```
{
  "msg" : "0_on",
  "sender" : {
    "type" : "device",
    "id" : "A",
    "room" : "Living room",
    "deviceType" : {
      "name" : "LightType"
    },
    "consumption" : 5
  }
}
```

Le topic utilizzate per comunicare direttamente dai/con i devices sono formate da una stringa composta da più parti separate dal carattere slash ('/'). La prima parte indica la stanza del device, la seconda il tipo di device, la terza il suo id e la quarta se la topic è usata per messaggi in entrata o in uscita. Esempi di topic sono:

- "Living room/LightType/A/To": usata per inviare messaggi al Device A;
- "Living room/LightType/A/From": usata dal Device A per inviare messaggi, su questa topic è in ascolto il Coordinatore una volta avvenuta la registrazione;
- "registration": usata dal Device per registrarsi presso il Coordinatore;
- "broadcast": usata dal Coordinatore per inviare un messaggio in broadcast a tutti i Devices;
- "update": usata dal Device per inviare una conferma di avvenuto aggiornamento in seguito a comando;
- "log": usato per trasmettere informazioni riguardanti l'accensione/spegnimento dei device.

Capitolo 6

Implementazione

6.1 Coordinatore

6.1.1 Profili Preimpostati

Un profilo preimpostato permette al programmatore di definire il comportamento del nostro sistema impostando una serie di funzioni, esse saranno composte da una serie di case match applicati ad ogni dispositivo mediante una funzione applyCommand. In particolare un profilo è composto da una serie di comandi *initialRoutine*, eseguiti non appena il profilo viene attivato e da una serie di funzioni ****NotificationCommands* che permettano di definire come il sistema debba reagire ad una notifica inviata da un sensore.

```
trait BasicProfile extends Profile {
  val initialRoutine: Device => Unit
  def thermometerNotificationCommands(room: String,value: Double):
  ↪  Device => Unit
  def hygrometerNotificationCommands(room: String,value: Double):
  ↪  Device => Unit
  ...

  def applyCommand(command: Device => Unit): Unit = {
      for (device <- Coordinator.getDevices) {
        command(device)
      }
  }

  override def onActivation(): Unit = applyCommand(initialRoutine)

  override def onThermometerNotification(room: String, value:
  ↪  Double): Unit =
  ↪  applyCommand(thermometerNotificationCommands(room,value))
```

```
  ...
}
```

Un esempio di profilo preimpostato è il profilo *NIGHT* che, una volta attivato, spegne tutti i dispositivi nel sistema tranne il climatizzatore ed il deumidificatore (impostandone anche i relativi valori) e, nel caso in cui venga rilevata una presenza in una determinata stanza, vengono accese le luci appartenenti a quest'ultima (ad una luminosità bassa) e spegnendole nel caso in cui il sensore non rileverà più alcuna presenza.

```
private case object NIGHT extends BasicProfile  {

    override val name: String = "NIGHT"
    override val description: String = "Night Profile"

    /** This profile turns off each device but the Air Conditioner
     ↪  and Dehumidifier, also closes Shutters **/
    override val initialRoutine: Device => Unit = {
      case device: AssociableDevice if device.deviceType ==
       ↪  ShutterType => Coordinator.publish(device,
       ↪  CommandMsg(cmd = Msg.close));
      case device: AssociableDevice if device.deviceType ==
       ↪  AirConditionerType => Coordinator.publish(device,
       ↪  CommandMsg(cmd = Msg.on)); Coordinator.publish(device,
       ↪  CommandMsg(Msg.nullCommandId, Msg.setTemperature, 25))
      case device: AssociableDevice if device.deviceType ==
       ↪  DehumidifierType => Coordinator.publish(device,
       ↪  CommandMsg(cmd = Msg.on)); Coordinator.publish(device,
       ↪  CommandMsg(Msg.nullCommandId, Msg.setHumidity, 40))
      case device: AssociableDevice if
       ↪  !DeviceType.isSensor(device.deviceType) =>
       ↪  Coordinator.publish(device, CommandMsg(cmd = Msg.off))
      case _ =>
    }

    override def thermometerNotificationCommands(room: String,
     ↪  value: Double): Device => Unit = _ => ()

    ...

    /** If someone is walking in the dark we bright it up a
     ↪  little, turns off when everyone leaves **/
    override def motionSensorNotificationCommands(room: String,
     ↪  value: Boolean): Device => Unit = {
```

```
      case device: AssociableDevice if value && device.room ==
       ↪  room && device.deviceType == LightType =>
        Coordinator.publish(device, CommandMsg(cmd = Msg.on))
        Coordinator.publish(device, CommandMsg(Msg.nullCommandId,
         ↪  Msg.setIntensity, 30))
      case device: AssociableDevice if !value && device.room ==
       ↪  room && device.deviceType == LightType =>
        Coordinator.publish(device, CommandMsg(cmd = Msg.off))
      case _ =>
    }
  }
```

6.1.2 Profili Custom

Per l'implementazione dei profili custom è stata realizzata una classe CustomProfile.
In questo caso la scelta delle varie regole in base alle quali applicare i comandi, nonché l'istanziamento, avvengono a runtime. Per quanto riguarda le regole, inizialmente viene generato un set di comandi chiamato *onActivation*, relativi ai dispositivi scelti dall'utente mediante la GUI. Tramite poi una funzione chiamata *generateCommandSet*, dato un set di dispositivi e comandi viene generato un Set di funzioni Device => Unit che, una volta richiamato, confronta l'ID di ogni device registrato all'interno del coordinatore con la scelta dell'utente, che in caso di match applica il relativo comando.

```
//Set of device and command
  def generateCommandSet(commands: Set[(Device,CommandMsg)]):
   ↪  Set[Device => Unit] = {
    var result: Set[Device => Unit] = Set.empty

    for(command <- commands) {
      val device: Device = command._1
      val message: CommandMsg = command._2

      result += {
        _.id match {
          case t if t == device.id =>
           ↪  Coordinator.publish(device.asInstanceOf[AssociableDevice],
           ↪  message)
          case _ => null
        }
      }
    }
    result
  }
```

Nel caso dei comandi applicati in risposta ai valori ricevuti dai sensori che registrano un valore numerico (temperatura, umidità, luce solare), viene utilizzata sempre la stessa funzione Mappata però all'interno di una Map[(String, Double) => Boolean, Set[Device => Unit]].
Questa mappa possiede come chiave una funzione che, dato un valore String (la stanza della rilevazione) ed un Double (il valore rilevato), restituisce un valore Booleano, che nel caso in cui corrisponda a true, allora vorrà dire che un criterio specificato dall'utente è stato soddisfatto e pertanto il relativo Set di comandi verrà applicato.

Funzione usata per generare i controlli:

```
def generateCheckFunction(symbol: String, value: Double,
 ↪  consideredRoom: String):
(String, Double) => Boolean = symbol match {
    case "=" => {
      case (room, double) if double == value && room ==
       ↪  consideredRoom => true
      case _ => false
    }
    case "<" => ....
}
```

Questa mappa è stata resa necessaria cosicché l'utente possa scegliere diversi set di comandi applicabili in base ai diversi valori ricevuti, ad esempio l'utente vuole che vengano eseguito un certo Set di comandi in reazione ad un valore di temperatura superiore a 32 rilevato nel salotto ed un altro Set di comandi per un valore di temperatura inferiore a 24 in garage. Nel caso di sensori in cui non è prevista la registrazione di valori numerici (es. movimento), è stata implementata una Mappa che come chiave utilizza unicamente un valore String, corrispondente alla stanza in cui il sensore ha rilevato un valore: *Map[String, Set[Device => Unit]]*.

6.2 Device

Nella gerarchia dei device ciascuno di essi deriva da un trait principale "Device" il quale implementa i tratti comuni a tutti come l'id, tipologia o consumo. Dal trait principale estendono vari trait:

- AssociableDevice: trait principale per i dispositivi connessi mediante MQTT. Implementa tutti i metodi e funzioni necessari alla comunicazione in rete, in particolare il metodo onMessageReceived gestisce i vari messaggi ricevuti in input. Nel caso in cui un messaggio sia specifico per una determinata tipologia di dispositivo, viene richiamato il metodo handleDeviceSpecificMessage il quale permetterà di gestirlo correttamente.

```
    /** Handles the received messages
     *
     * @param topic the topic the message arrived on
     * @param msg the message itself
     */
    def onMessageReceived(topic: String, msg: String): Unit = {
      lazy val message: String = getMessageFromMsg(msg)
      topic match {
        case t if t == subTopic => message match {
          case m if m == Msg.regSuccess => _registered = true;
           ↪  registrationPromise.success(() => Unit)
          case m if m == Msg.disconnect => disconnect
          case m if CommandMsg.fromString(m).command == Msg.on
           ↪  => if(turnOn()) {sendConfirmUpdate(message);
           ↪  sendLogMsg(Msg.on)}
          case m if CommandMsg.fromString(m).command == Msg.off
           ↪  => if(turnOff()) {sendConfirmUpdate(message);
           ↪  sendLogMsg(Msg.off)}
          case _ => if
           ↪  (handleDeviceSpecificMessage(CommandMsg.fromString(message)))
           ↪  sendConfirmUpdate(message)
        }
        case t if t == broadcastTopic => message match {
          case m if m == Msg.disconnected =>
            turnOff()
            _registered = false
          case _ => this.errUnexpected(UnexpectedMessage,
           ↪  message)
        }
        case _ => this.errUnexpected(UnexpectedTopic, topic)
      }
    }
```

- ChangeableValue: trait da usare per mixin nei device che implementano un valore con minimo/massimo che può venire cambiato.

```
    /** Represents a devices with a changing value **/
  sealed trait ChangeableValue extends Device {
    //min, max value for the intensity
    def minValue : Int
    def maxValue : Int
    var value : Int
```

```
    private def _mapValue: Int => Int =
     ↪  ValueChecker(minValue,maxValue)(_)

    def setValue(newValue: Int): Boolean = { value =
     ↪  _mapValue(newValue); true}
    def getValue: Int = value
  }
```

- MutableExtras: trait da usare per mixin che implementa un set di funzioni extra necessari ad esempio ad una lavatrice o ad un forno.
- SensorAssociableDevice: tipologia di Device che rappresenta un sensore, implementa delle funzioni aggiuntive per notificare il coordinatore in caso di nuova rilevazione (solamente in caso di variazione significativa).

6.2.1 DeviceType

Per gestire le varie tipologie di Device si utilizza un trait di supporto DeviceType, richiesto principalmente dai profili preimpostati per effettuare modifiche su tutti i dispositivi appartenenti ad una determinata tipologia. L'oggetto DeviceType offre anche alcune funzioni utili per determinare la natura di un Device trattando il DeviceType similmente ad un Enum e sfruttando delle modifiche alla classe mediante un Pimp my Library, oltre che fare da builder tramite il metodo apply.

```
/** Object of DeviceTypes with Utils and a Factory **/
object DeviceType {
  def listTypes: Set[DeviceType] = Set(LightType,
   ↪  AirConditionerType, DehumidifierType, ShutterType,
   ↪  BoilerType, TvType, WashingMachineType, DishWasherType,
   ↪  OvenType, StereoSystemType)
  def sensorTypes: Set[DeviceType] = Set(ThermometerType,
   ↪  HygrometerType, MotionSensorType, PhotometerType)

  //an Enum-Like approach to check the name of case objects
  def isSensor(senType: String): Boolean =
   ↪  sensorTypes.findSimpleClassName(senType)
  def isSensor(deviceType: DeviceType): Boolean =
   ↪  sensorTypes.findSimpleClassName(deviceType.getSimpleClassName)

  def isDevice(devType: String): Boolean =
   ↪  listTypes.findSimpleClassName(devType)
  def isDevice(deviceType: DeviceType): Boolean =
   ↪  listTypes.findSimpleClassName(deviceType.getSimpleClassName)
```

```
  def apply(devType: String): DeviceType = (listTypes ++
  ↪  sensorTypes).find(_.getSimpleClassName == devType) match {
    case Some(t) => t
    case _ => this.errUnexpected(UnexpectedDeviceType, devType)
  }
}
```

6.3 GUI

6.3.1 Sensori e simulazione

La modalità per cui GUI differenzia il modo in cui gestisce i sensori rispetto ai restanti device è data data dall'oggetto *PrintDevicePane*, la cui funzione apply è la seguente.

```
  def apply(device: Device) : GUIDevice = device.deviceType
  ↪  match{
    case AirConditionerType =>
    ↪  AirConditionerPane(AirConditioner(device.name,device.room))
       /**...All other cases referring each to a device
       ↪  type...*/

    //Sensors
    case ThermometerType =>
    ↪  ThermometerPane(device.asInstanceOf[SimulatedThermometer])
       /**...All other cases referring each to a sensor
       ↪  type...*/
    case _ => this.errUnexpected(UnexpectedDeviceType,
    ↪  device.deviceType.toString)
  }
```

Questa funzione viene chiamata ogni qual volta che un device deve essere aggiunto alla GUI:

- in fase di inizializzazione: i device vengono presi dal coordinatore e dunque sono un riferimento ai device connessi;
- tramite input utente: il device viene registrato tramite un tool apposito denominato RegisterDevice.

In entrambi i casi, come si può vedere nel codice qui sopra, per qualunque device che non sia un sensore, il valore di ritorno è un *GUIDevice* contenente all'interno una copia del device passato in input; al contrario, i *GUIDevice* inerenti ai sensori contengono al loro interno un riferimento diretto al termometro connesso(*.asInstanceOf[SimulatedThermometer]*).

6.3.2 Gestione aspetti non istantanei

All'interno del sistema vi sono aspetti, in particolare funzioni, che potrebbero venire completate nel tempo e non istantaneamente, così come ci sono procedure che prima di terminare necessitano che ne terminino altre; si prenda il caso più eclatante della modifica di una proprietà di un device tramite interfaccia grafica: tale modifica deve essere comunicata al coordinatore che a sua volta deve effettuare una publish e ottenere una risposta dal device di conferma update. E' necessario dunque, per garantire integrità, che GUI, ogni qual volta comunichi al coordinatore una modifica, attenda che il coordinatore riceva la conferma dell'update da parte del device prima di aggiornare l'interfaccia grafica col nuovo valore; togliendo questo meccanismo, in caso di problemi di rete, si sarebbe verificata una collisione tra lo stato del dispositivo connesso e la sua copia mantenuta in GUI.

```
def userUpdate(devName : String,cmdMsg :String,newValue:String):
↪  Future[Unit] = {
  val p = Promise[Unit]
  Coordinator.sendUpdate(devName, cmdMsg, newValue).onComplete {
    case Success(_) => setVal(newValue); p.success(() => Unit);
    case _ => Dialog.showMessage(title ="Update Error",message =
    ↪  "Something wrong happened while trying to update a
    ↪  device",messageType = Message.Error);  p.failure(_)
  }
  p.future
}
```

Ogni *promise* generata da una richiesta lato utente, viene temporaneamente associata ad un id (che equivale all'id della richiesta oggetto della publish del coordinatore) e salvata in una lista. Quando poi il dispositivo connesso confermerà l'update con una publish contenente l'id, tale promise verrà completata.

```
object RequestHandler {
  private var updateRequests : Map[Int,Promise[Unit]]= Map.empty
  private var nextNumber : Int = 0

  def addRequest(newRequest :Promise[Unit]): Int = {
    nextNumber += 1
    updateRequests += (nextNumber -> newRequest); nextNumber
  }

  def handleRequest(id : Int): Unit = {
    updateRequests(id).success(() => Unit); updateRequests -= id
  }
}
```

Un altro aspetto gestito tramite l'utilizzo di promise/future è quello della registrazione dei device, che potrebbe non essere istantanea e soprattutto potrebbe fallire.

6.3.3 Login

Un'ultima funzionalità degna di menzione è quella del login: come definito nei requisiti, considerando che lo scopo di questo sistema sarebbe quello di gestire i dispositivi interni ad una casa, è necessario fornire almeno un minimo di sicurezza e prevenire accessi non desiderati all'interno dell'applicazione. Per questo è stata costruita una semplice funzionalità di login che utilizza PBKDF2 [6] come algoritmo di hashing e si occupa di registrare, salvare le credenziali dell'utente all'interno della sua cartella *user.home* e verificare, al momento della presentazione, che le credenziali fornite da un utente siano corrette e possa essere garantito l'accesso al sistema.

6.4 MQTT

L'indirizzo del broker MQTT differisce da ambiente di sviluppo a quello di produzione, infatti durante le fasi di sviluppo il broker utilizzato è installato in locale, mentre in produzione questo è lanciato in un container Docker dedicato. La soluzione adottata per risolvere questa discrepanza consiste in un retry della connessione:

1. In un primo momento, Devices e Coordinatore tentano di connettersi al broker fornito dal servizio Docker all'indirizzo *tcp://mosquitto-service:1883*;
2. In caso di fallimento, viene tentata un'ulteriore connessione all'eventuale broker locale presente all'indirizzo *tcp://localhost:1883*.

In questo modo il codice dell'applicazione non varia a seconda del modo in cui è lanciata: funziona sia con fat-jar + Mosquitto Broker in locale che tramite docker-compose.

6.5 JSON

Ogni messaggio JSON inviato è composto da body e sender del messaggio. E' interessante notare come in Scala sia possibile configurare write e read di interi oggetti in/da stringa JSON. Questa caratteristica è stata sfruttata, per mezzo di *implicit*, per la scrittura e lettura del mittente. Infatti il sender di un messaggio è per costruzione un JSONSender ed il compilatore automaticamente cerca il write/read più appropriato per risolvere la chiamata a metodo. Di seguito ne viene mostrato un esempio:

```
def getMsg(message: String, sender: JSONSender): String = {
    if (message == null || sender == null) return null
    JsObject(
      Seq(
        msgField -> JsString(message),
```

```
            senderField -> Json.toJson(sender),
        )
    ).toString()
}
private implicit val jsonSenderWrites: Writes[JSONSender] = {
    case sender@s if s.isInstanceOf[AssociableDevice] =>
     ↪  deviceWrites.writes(sender.asInstanceOf[AssociableDevice])
    case s if s == Coordinator =>
     ↪  coordinatorWrites.writes(Coordinator)
}

private implicit val deviceWrites: Writes[AssociableDevice] =
 ↪  (device: AssociableDevice) => Json.obj(
    typeField -> device.senderType._type,
    idField -> device.id,
    roomField -> device.room,
    deviceTypeField -> device.deviceType,
    consumptionField -> device.consumption,
)

private implicit val deviceTypeWrites: Writes[DeviceType] =
 ↪  (deviceType: DeviceType) => Json.obj(
    nameField -> deviceType.getSimpleClassName
)
```

In questo caso, la chiamata *Json.toJson(sender)* accetta in ingresso un *implicit tjs: Writes[T]*, che viene risolto con il *val jsonSenderWrites*, questo a sua volta redireziona la chiamata al *val deviceWrites* che si occupa di fare la write di un AssociableDevice. Nel farlo, mappa *deviceTypeField -> device.deviceType* che viene automaticamente risolto grazie al *val deviceTypeWrites*. Analogamente avviene per quanto riguarda le reads, che sfruttano il metodo apply degli objects JSONSender per ricreare l'oggetto a partire dai campi contenuti nel JSON message.

6.6 Gradle-Docker-Travis Configs

Un aspetto molto importante per permettere di monitorare ed automatizzare alcune fasi del ciclo di produzione del software (build, test, deploy) è l'integrazione dei servizi Gradle, Docker Compose e Travis CI. Le difficoltà principali incontrate sono legate all'utilizzo di Docker Compose per creare e lanciare l'immagine di un'applicazione con interfaccia grafica. La soluzione adottata è di seguito riportata.

6.6.1 Gradle

Il file di configurazione *build.gradle* è stato modificato per includere il plugin in grado di generare un fat-jar dell'applicazione e dei tasks gradle in grado di permettere selettivamente l'esecuzione di tutti i tests o solo di quelli che non richiedono l'utilizzo del broker mosquitto attivo.

6.6.2 Docker e Docker Compose

Un DockerFile è presente nella cartella del progetto per permettere una multistage build: quando viene chiamato dal Docker Compose questo compila l'applicazione e produce un fat-jar. Inoltre installa nell'ambiente virtuale Docker (cioè nell'immagine che verrà creata) i software necessari all'esecuzione dell'applicazione in un secondo momento, come jdk8 e VNC.

6.6.3 Travis CI

Il file di Travis allegato permette una build automatica dei sorgenti e l'esecuzione dei tests. Questo include l'utilizzo dei servizi docker (per lanciare il broker mosquitto) e xvfb (per permettere l'esecuzione di tests che necessitano di un monitor, fornisce un monitor virtuale). In caso di errore Travis informa l'owner del repository, altrimenti lancia automaticamente i comandi *docker-compose build* e *docker-compose push* che permettono di creare l'immagine dell'applicazione e di caricarla su un Docker Hub accessibile dall'esterno.

6.6.4 Docker Compose

L'utente finale deve riportare la cartella contenente i files di configurazione del broker mosquitto (contenenti anche le coppie username/password hash valide per connettersi al broker) ed il docker-compose file fornito nella cartella *releases*. Grazie a questi, il comando *docker-compose up* scarica automaticamente le immagini del broker e dell'applicazione e avvia quest'ultima. Per interagire tramite interfaccia grafica occorre infine collegarsi con VNC all'indirizzo 127.0.0.1:5900.

Capitolo 7

Installazione ed Esecuzione

E' possibile installare ed eseguire l'applicazione seguendo tre procedure alternative e distinte.

La prima richiede l'utilizzo di Docker Compose e di VNC Viewer:

1. Creare una nuova cartella, qui chiamata a titolo di esempio *HOMEApp*
2. Copiare il file */releases/docker-compose.yml* in *HOMEApp*
3. Copiare l'intera cartella */releases/mosquitto* in *HOMEApp*
4. Lanciare da riga di comando l'istruzione *docker-compose up* a partire dal path di *HOMEApp*
5. Al completamento delle operazioni, connettersi tramite VNC Viewer all'indirizzo *127.0.0.1:5900*

La seconda procedura richiede l'installazione preventiva di Gradle, jdk8 e Mosquitto [3]:

1. Creare una nuova cartella, qui chiamata a titolo di esempio *HOMEApp*
2. Copiare l'intera cartella */releases/mosquitto* in *HOMEApp*
3. Lanciare da riga di comando l'istruzione *mosquitto -c mosquitto/config/mosquitto.conf -v* a partire dal path di *HOMEApp*
4. Scaricare i sorgenti dell'applicazione e creare i jar relativi con l'istruzione *gradle assemble*
5. Lanciare il jar eseguibile generato in */build/libs/PPS-19-HOME-all.jar*

La terza procedura è una scorciatoia della seconda e richiede l'installazione preventiva di jdk8 e Mosquitto [3]:

1. Creare una nuova cartella, qui chiamata a titolo di esempio *HOMEApp*
2. Copiare l'intera cartella */releases/mosquitto* in *HOMEApp*

3. Lanciare da riga di comando l'istruzione *mosquitto -c mosquitto/config/mosquitto.conf -v* a partire dal path di *HOMEApp*

4. Lanciare il jar eseguibile presente in */releases/PPS-19-HOME-all.jar*

Capitolo 8

Retroprospettiva

Restrospettiva (descrizione finale dettagliata dell'andamento dello sviluppo, del backlog, delle iterazioni; commenti finali)

8.1 Konrad Gomulka

In generale ritengo questo progetto una gran bella esperienza, programmare con scala è stato indubbiamente difficile ma al contempo soddisfacente. Le potenzialità e "l'eleganza" di questo linguaggio sono diventate sempre più chiare man mano che lo sviluppo andava avanti così come il crescente interesse nello sviluppare in tale linguaggio.
Se costretto a denotare dei problemi nello sviluppo l'unica cosa che mi viene in mente probabilmente è, in alcuni casi, il non aver priorizzato correttamente alcune delle funzionalità principali eclissandole con altre, mentre l'unico rimpianto rimasto a fine progetto è di non averlo continuato date le innumerevoli possibilità di ulteriore sviluppo e ampliamento.

8.2 Manuele Pasini

Le sensazione arrivati alla fine di questo progetto sono un po' contrastanti, probabilmente frutto anche del periodo in cui è capitato: ho apprezzato molto Scala (col tempo) e soprattutto l'idea di tentare un approccio diverso da quello che dalla prima superiore ad oggi mi hanno insegnato; ho tuttavia trovato l'apprendimento difficile, ho problemi di attenzione e il seguire le lezioni da casa si è rivelato per me deleterio, mi sono trovato ad inizio progetto senza la base che avrei voluto avere, di conseguenza si sono allungati i tempi di realizzazione però ormai *alea iacta est* e sono comunque complessivamente soddisfatto di quanto prodotto come gruppo e come singolo. E' stata molto interessante anche l'esperienza, o quantomeno il tentativo di esperienza, con SCRUM, che ho trovato una valida alternativa a come solitamente organizziamo il lavoro, mi piacerebbe approfondire maggiormente questa metodologia che per quanto abbiamo provato ad applicare al meglio, ha comunque causato qualche "rallentamento" (per il semplice fatto che richiede un'organizzazione maggiore e migliore rispetto a come siamo stati abituati a lavorare fino ad ora). Per quanto riguarda il progetto in se, sono soddisfatto del prodotto finale anche se, viste le possibilità

di espansione di questo, sembra sempre di lasciare un lavoro a metà. Sono soddisfatto anche del gruppo, considerando che su quattro componenti solo due avevano già lavorato assieme, c'è sempre stata una buona comunicazione e tutte le problematiche sono state risolte senza problemi. Unica pecca di tutto il progetto, credo sia stata un po' tralasciata la parte di progettazione, che ha portato a qualche improvvisazione e conseguente difficoltà.

8.3 Andrea Sperandio

Sono piuttosto soddisfatto dei risultati ottenuti attraverso il progetto. Personalmente sono più orientato agli aspetti che riguardano la gestione del team, del ciclo di produzione e dell'integrazione continua, quindi ho particolarmente apprezzato le opportunità di mettermi alla prova con l'utilizzo di Trello, la metodologia SCRUM, GitFlow, Gradle, Docker e Travis CI. Con il progredire del progetto ho anche apprezzato la potenza e le potenzialità del linguaggio Scala, nonostante ciò che mi ha colpito ed interessato maggiormente sia stata la componente di interazione e collaborazione umana conseguente all'approccio di sviluppo utilizzato. Un altro aspetto molto gradito è legato alla scrittura dei tests: infatti questi hanno accompagnato l'intero sviluppo dell'applicazione e sono risultati particolarmente utili durante le fasi di refactoring. Dovendo fare un'analisi a tutto tondo, tra gli aspetti non proprio riusciti mi sento di includere la mancata progettazione a monte dell'integrazione tra GUI ed il resto dell'applicazione ed un focus non sempre corretto e bilanciato tra requisiti obbligatori ed opzionali, che ha portato ad un allungamento dei tempi di realizzazione.

8.4 Luca Salvigni

Devo dire che nonostante la notevole difficoltà nella realizzazione del progetto, sono rimasto piuttosto contento e soddisfatto del traguardo che abbiamo raggiunto. Una delle situazioni molto scomode in cui mi sono trovato è stata sicuramente quella di non avere avuto sin da subito una buona base per quanto riguardasse Scala. Questo problema è dovuto, dal mio punto di vista, molto dalle lezioni online, non riuscendomi a concentrare o ad immergermi nelle lezioni come magari riuscivo a fare in una lezione in aula. Purtroppo ciò, come detto anche dal mio compagno Manuele, ha portato ad un rallentamento per la realizzazione del progetto stesso. Detto questo però con il passare dei giorni devo dire che Scala mi è davvero piaciuto molto, sia perché lo trovo un linguaggio molto più intuitivo rispetto al solito Java, sia perché comunque essendo un nuovo tipo di linguaggio c'è molta più curiosità nel conoscere le sue potenzialità. SCRUM è stata una metodologia di lavoro davvero molto interessante e innovativa, non avendo mai lavorato con così tanta organizzazione. Pensavamo di poter consegnare il progetto molto prima della deadline prefissata, però a causa di una progettazione non svolta nel migliore dei modi, ci siamo ritrovati ad un mese dall'inizio che non avevamo ancora svolto metà del lavoro. Tutto sommato devo dire che però gli errori sono fondamentali per imparare e soprattutto per cercare di non commetterli in futuro, aumentando di conseguenza il proprio bagaglio di esperienza.

Bibliografia

[1] Lukas Rytz. Scala-swing, scala wrappers for java's swing api. https://github.com/scala/scala-swing.

[2] Information technology. message queuing telemetry transport (MQTT) v3.1.1.

[3] Eclipse Foundation. Eclipse mosquitto ™. https://mosquitto.org/.

[4] Daniele Alberti. Primi passi con mqtt e arduino. https://www.danielealberti.it/2018/09/primi-passi-con-mqtt-e-arduino.html.

[5] The HiveMQ Team. Mqtt essentials. https://www.hivemq.com/mqtt-essentials/.

[6] B. Kaliski and A. Rusch. PKCS #5: Password-based cryptography specification version 2.1. Technical report, doi.org, January 2017.