SmartHome
Ingegneria del Software

Cavaleri Fabio - 851587
Lanzani Federico - 852273
Mazzoletti Lorenzo - 851940
Pirolo Davide - 852002

A.A 2021-2022

Indice

Indice delle immagini

Capitolo 1

Introduzione

Il progetto SmartHome si colloca all'interno di un contesto molto ampio e molto complesso da sviluppare per intero.
Per questo motivo, sulla base delle funzionalità offerte dal framework PatrIot[1], abbiamo considerato alcune aree del contesto come già realizzate in modo da poterle utilizzare come base per lo sviluppo delle restanti funzionalità del sistema.
In particolare ci siamo concentrati sullo sviluppo di:

1. Logica del sistema
2. Middleware per la comunicazione tra device fisici e logica
3. Interfaccia utente

Le dinamiche e la comunicazione tra queste entità saranno mostrate in maniera più chiara e dettagliata nei capitoli successivi.

1.1 Logica del sistema

Il compito principale della logica è la gestione autonoma del sistema, verificando la presenza di anomalie e individuando le azioni necessarie per risolverle.
In particolare la logica si occupa di:

- Analizzare le rilevazioni effettuate dai device fisici (es. sensori)
- Riconoscere, sulla base delle rilevazioni, eventuali problematiche all'interno del sistema
- Identificare le azioni necessarie alla risoluzione dei problemi
- Verificare che le azioni identificate non siano in conflitto con altre operazioni in esecuzione nel sistema

1.2 Middleware

Questo elemento dell'architettura fa da intermezzo tra i dispositivi fisici del sistema e la logica applicativa.
In particolare il middleware si occupa di:

- Ricevere gli input dai device fisici del sistema (es. sensori)
- Inviare gli input dei device fisici alla logica per elaborarli
- *Ricevere gli input inviati dall'utente*
- *Inviare le azioni richieste dall'utente alla logica per analizzarle*
- Comunicare con la base di dati
- Comunicare ai device fisici (attuatori) le azioni richieste dalla logica del sistema *o dall'utente*

[1] Dopo un'attenta analisi del framework iCasa Framework, suggerito nel testo del progetto, abbiamo deciso di utilizzare soluzioni alternative poiché quella proposta è risultata essere poco documentata e non supportata da diversi anni.

1.3 Interfaccia utente

Attraverso questo elemento dell'architettura l'utente può interfacciarsi con il sistema per:

- Verificare lo stato dei dispositivi presenti all'interno dell'abitazione
- *Richiedere l'utilizzo di un attuatore*

1.4 Aree di interesse

Insieme alla scelta di quali elementi dell'architettura realizzare, abbiamo individuato le aree di interesse nelle quali il sistema sarà in grado di operare all'interno della casa:

- Controllo dell'illuminazione
  - Gestione delle luci
- Controllo della temperatura
  - Gestione dell'impianto di riscaldamento
  - Gestione dell'impianto di condizionamento
- Controllo della pulizia
  - Gestione dei dispositivi per la pulizia della casa
- Controllo della sicurezza
  - Controllo anti-intrusione
  - Controllo pericoli
    - Gestione delle finestre

Nota

L'imprevista durata prolungata della fase iniziale del progetto, le cui motivazioni sono approfondite nell'Appendice A, hanno limitato le funzionalità del sistema che siamo stati in grado di sviluppare.
Inizialmente avevamo programmato di realizzare anche la gestione delle tapparelle e delle porte, insieme alla possibilità per dell'utente di richiedere l'esecuzione di azioni da parte degli attuatori.
Nei capitoli successivi i riferimenti a queste funzionalità saranno indicati in *corsivo*.

Capitolo 2

Analisi dei Requisiti

2.1 Introduzione

Data la complessità del contesto a cui appartiene il progetto ci siamo concentrati sullo sviluppo della parte principale dell'architettura e delle funzionalità appartenenti alle aree di interesse principali.
Vista l'assenza di uno stakeholder con cui interagire e la poca conoscenza del mondo delle smart home, abbiamo seguito alcune linee guida, descritte in un articolo pubblicato su *International Journal of Engineering Research and Applications*[1], per l'ideazione del sistema e della sua architettura.
Abbiamo quindi formulato i requisiti immaginando degli scenari di utilizzo, in modo da rispecchiare le caratteristiche fondamentali utili per una realizzazione completa delle funzionalità appartenenti alle aree di interesse scelte nella fase di ideazione.

2.2 Requisiti Funzionali

- Il sistema deve essere in grado di ricevere le rilevazioni effettuate dai dispositivi fisici[1]
- L'utente deve avere la possibilità di visualizzare lo stato dei dispositivi fisici[1] della casa

2.2.1 Gestione dell'impianto di illuminazione

- Il sistema deve regolare l'illuminazione di una stanza a seconda della presenza o meno di persone al suo interno
- *L'utente deve poter chiedere al sistema di accendere o spegnere la la luce in una stanza*
- Il sistema deve adattare la regolazione delle luci al ciclo giorno/notte

2.2.2 Gestione della temperatura

- *L'utente deve avere la possibilità di impostare una temperatura ideale che vuole mantenere all'interno della casa*
- Il sistema deve essere in grado di rilevare la temperatura di una stanza della casa
- Il sistema deve regolare l'impianto di riscaldamento di una stanza a seconda della temperatura
- Il sistema deve regolare l'impianto di condizionamento di una stanza a seconda della temperatura
- *L'utente deve avere la possibilità di attivare/disattivare l'impianto di riscaldamento di una stanza.*
- *L'utente deve avere la possibilità di attivare/disattivare l'impianto di condizionamento di una stanza.*
- Il sistema deve adattare il proprio comportamento alle stagioni

[1] Le categorie ed i tipi di dispositivi fisici gestiti dal sistema sono descritti in maniera più approfondita nei capitoli successivi della documentazione

2.2.3 Gestione dei dispositivi per la pulizia della casa

- Il sistema deve essere in grado di avviare le operazioni di pulizia periodicamente
- *L'utente deve avere la possibilità di avviare le operazioni di pulizia*

2.2.4 Controllo anti-intrusione

- *L'utente deve avere la possibilità di attivare e disattivare il sistema di allarme*
- Il sistema deve essere in grado di rilevare intrusioni nell'abitazione
- Il sistema deve essere in grado di eseguire un procedura di emergenza in caso di intrusione

2.2.5 Controllo pericoli

- Il sistema deve essere in grado di rilevare la presenza di una fuga di gas
- Il sistema deve essere in grado di rilevare la presenza di fumo nell'abitazione
- Il sistema deve essere in grado di eseguire una procedura di emergenza se rileva fughe di gas o fumo

2.3 Requisiti non Funzionali

- Il sistema deve essere essere in grado di evitare conflitti tra le operazioni elaborate dalla logica o richieste dall'utente

Nota

Come descritto nel capitolo introduttivo, i riferimenti alle funzionalità progettate inizialmente ma non realizzate sono riportati in *corsivo*.

Capitolo 3

Casi d'Uso

3.1 Introduzione

Come già presentato nel capitolo precedente, la presenza del cliente nella fase di analisi è fondamentale per la riuscita di questa parte iniziale del progetto. I requisiti e i casi d'uso infatti devono essere modellati mantenendo il punto di vista dell'utente.
Per la formulazione dei casi d'uso abbiamo iniziato dalla definizione dei confini del sistema e degli attori; in questo modo, facendo riferimento ai requisiti funzionali realizzati nel capitolo precedete, abbiamo modellato i casi d'uso cercando di mantenere il punto di vista degli attori ed in particolare dell'utente finale.

3.2 Confini del Sistema

Durante la definizione dei confini del sistema è stato necessario iniziare la modellazione dell'architettura software del progetto, in modo da poter distinguere le parti del sistema che fungono da attori e quelle che invece stanno all'interno dei confini.
In particolare nei confini rientra quella parte del sistema che abbiamo definito come logica nel capitolo introduttivo; ovvero quella sezione che si occupa dell'analisi delle rilevazioni, dell'identificazione dei problemi e della definizione delle azioni necessarie alla loro risoluzione.

3.3 Attori

3.3.1 Sistema

Con questo attore si intende quella parte del sistema che fa da tramite tra i dispositivi fisici e la logica descritta al punto precedente. Questa sezione del sistema è stata definita nel capitolo introduttivo come *Middleware per la comunicazione tra device fisici e logica.*
Questo attore può essere considerato come primario poiché utilizza la logica del sistema per offrire servizi all'utente.

3.3.2 Utente

Informalmente si potrebbe definire un utente come un abitante della casa a cui appartiene il sistema SmartHome.
Formalmente intendiamo come Utente un utilizzatore del sistema; ovvero qualcuno che, attraverso la user interface, può interagire con il sistema e richiederne i servizi.
Questo attore può assumere due ruoli differenti durante l'esecuzione dell'applicazione, quindi lo si può distinguere in due tipologie:

- **Primario:** *Quando richiede direttamente dei servizi al sistema interagendo attraverso l'interfaccia grafica*
- **Finale:** Quando il sistema, in autonomia, esegue delle azioni seguendo le preferenze dell'utente.

3.3.3 Dispositivi Fisici

Questo attore generalizza tutti i tipi dispositivi fisici che possono essere presenti all'interno di una smart home. In particolare abbiamo identificato queste due tipologie di device:

- **Sensori:** Sono quei dispositivi fisici che raccolgono dati effettuando delle rilevazioni e li inviano al sistema
- **Attuatori:** Sono quei dispositivi fisici di cui il sistema si serve per eseguire le azioni dettate dalla logica *o richieste dall'utente*

Anche questo attore può assumere ruoli diversi a seconda della tipologia di dispositivo di cui il sistema si avvale, in particolare può essere:

- **Primario:** Quando il sistema utilizza degli attuatori per eseguire delle azioni
- **Di supporto:** Quando il sistema utilizza dei sensori per effettuare delle rilevazioni

3.3.4 Database

Questo attore rappresenta la base di dati con cui il *Sistema* si interfaccia per il salvataggio e il recupero dei dati generati.
Possiamo considerare il *Database* come un attore di supporto poiché offre un servizio al sistema SmartHome.

3.4 Casi d'uso generali

- ***ControlloManuale:*** *L'attore Utente, attraverso la user interface, può richiedere al sistema di far eseguire ad un dispositivo fisico (attuatore) un'azione*
- **ControlloAutomatico:** L'attore Sistema può richiedere ad un dispositivo fisico (attuatore) di eseguire un'azione
- **VisualizzaStato:** L'attore Utente può visualizzare lo stato dei dispositivi fisici della casa
- ***AggiornaStato:*** *L'attore Sistema, in seguito ad un'azione dell'attore Utente attraverso il caso d'uso ControlloManuale, aggiorna lo stato del dispositivo fisico coinvolto.*
- **Esegui rilevazione:** L'attore Dispositivo Fisico (sensore), esegue una rilevazione che viene utilizzata nel caso d'uso RaccogliDati
- **RaccogliDati:** L'attore Sistema riceve le rilevazioni fatte dai Dispositivi Fisici(sensori) e le elabora nella logica del sistema
- **InviaDati:** L'attore Sistema riceve le rilevazioni fatte dai Dispositivi Fisici (sensori) e le invia al Database per salvarle
- **VerificaAzione:** L'attore Sistema verifica che l'azione elaborata dalla logica *o richiesta dall'Utente* non entri in conflitto con altre operazioni attualmente in esecuzione.

3.5 Gestione dell'impianto di illuminazione

- **ControlloLuci:** Gli attori Utente e Sistema, rispettivamente attraverso i casi d'uso ControlloManuale e ControlloAutomatico, possono accendere o spegnere un punto luce della casa.

3.6 Gestione della temperatura

- **ControlloRiscaldamento:** Gli attori Utente e Sistema, rispettivamente attraverso i casi d'uso ControlloManuale e ControlloAutomatico, possono controllare un dispositivo per il riscaldamento della casa.
- **ControlloCondizionamento:** Gli attori Utente e Sistema, rispettivamente attraverso i casi d'uso ControlloManuale e ControlloAutomatico, possono controllare dispositivo per il condizionamento della casa.

3.7 Gestione dei dispositivi per la pulizia della casa

- **ControlloPulizia:** Gli attori Utente e Sistema, rispettivamente attraverso i casi d'uso ControlloManuale e ControlloAutomatico, possono controllare i dispositivi per la pulizia della casa.

3.8 Controllo anti-intrusione

- **ControlloAllarme:** Gli attori Utente e Sistema, rispettivamente attraverso i casi d'uso ControlloManuale e ControlloAutomatico, possono controllare l'attivazione/disattivazione del sistema di allarme della casa.
- **ProceduraEmergenzaIntrusione:** L'attore Sistema, attraverso il casi d'uso ControlloAutomatico, può avviare la sequenza di azioni previste in caso di intrusione all'interno della casa.

3.9 Controllo pericoli

- **ProceduraEmergenzaIntrusione:** L'attore Sistema, attraverso il casi d'uso ControlloAutomatico, può avviare la sequenza di azioni previste in caso di pericoli all'interno della casa.

3.10 Precisazioni sui casi d'uso ControlloManuale e ControlloAutomatico

Per poter rappresentare le diverse modalità con cui si può interagire con i dispositivi fisici abbiamo inserito questi due casi d'uso base che vengono estesi da diversi casi che rappresentano i possibili scenari di utilizzo del sistema.
ControlloManuale e *ControlloAutomatico* includono il caso d'uso *VerificaAzione* che in entrambi gli scenari si occupa di controllare la compatibilità dell'azione richiesta con quelle attualmente in esecuzione nel sistema.

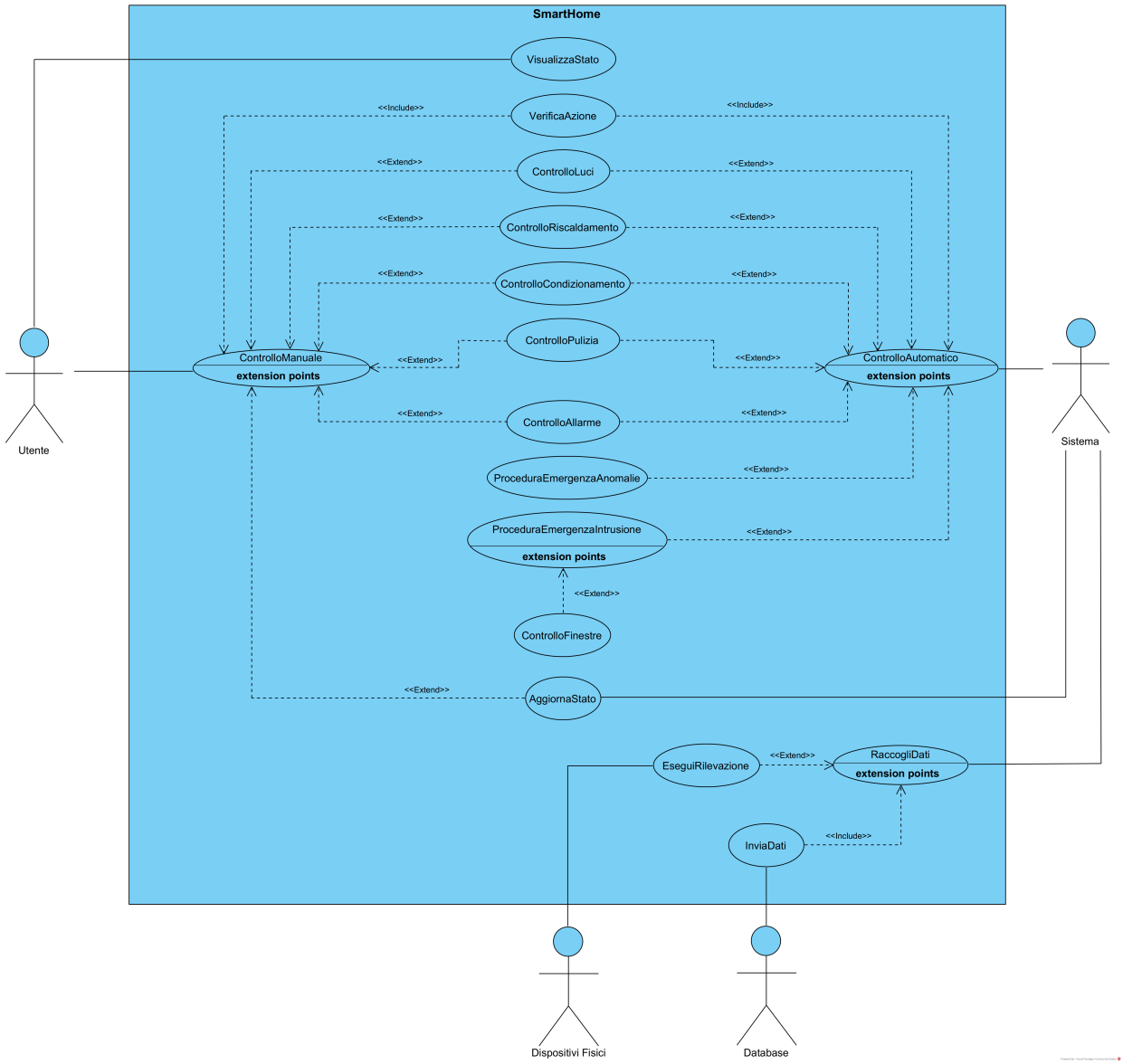

Immagine 3.1: Diagramma dei casi d'uso

Capitolo 4

Modello di Dominio

4.1 Introduzione

Gli elementi appartenenti alla fase di modellazione del diagramma delle classi a livello di dominio poggiano le loro radici negli elaborati prodotti durante le fasi precedenti del processo di sviluppo: ideazione, analisi dei requisiti e dei casi d'uso.
Questi elaborati infatti permettono di avere un panorama chiaro del dominio di cui fa parte il progetto, garantendo un rapido ed efficace riconoscimento delle classi concettuali, dei lori attributi e delle associazioni del diagramma.

4.2 Classi Concettuali

4.2.1 Utente

Questa classe concettuale rappresenta l'omonimo attore che abbiamo identificato nel capitolo precedente, ovvero "*qualcuno che, attraverso l'interfaccia grafica, può interagire con il sistema e richiederne i servizi*".

Attributi

- **userId:** Identificativo che l'utente utilizza per accedere al sistema
- **password:** Password che, insieme all'*userId*, l'utente utilizza per accedere al sistema

4.2.2 Middleware

Questa classe concettuale rappresenta l'attore che nel capitolo precedente abbiamo identificato con il nome *Sistema*, ovvero "*quel livello dell'architettura che fa da tramite tra i dispositivi fisici e la logica*".

4.2.3 Logica

Questa classe concettuale rappresenta quel livello dell'architettura che abbiamo definito come *Logica* nel capitolo introduttivo; ovvero "*quella sezione che si occupa dell'analisi delle rilevazioni, dell'identificazione dei problemi e della definizione delle azioni necessarie alla loro risoluzione*".

4.2.4 Dispositivo

Questa classe concettuale rappresenta l'attore che nel capitolo precedente abbiamo identificato con il nome *Dispositivo Fisico*, ovvero "*tutti i device fisici che possono essere presenti all'interno di una smart home (Sensori o Attuatori)*".

Attributi

- **id:** A ciascun dispositivo viene associato un identificativo univoco all'interno del sistema SmartHome.

4.2.5 Stanza

Questa classe concettuale permette di associare ogni Dispositivo alla stanza dell'abitazione in cui si trova. In questo modo il sistema è in grado di identificare in maniera più dettagliata le anomalie e di elaborare azioni più efficaci per la loro risoluzione.

4.2.6 Rilevazione

Questa classe concettuale rappresenta le rilevazioni effettuate da un Dispositivo (Sensore).
Questo elemento è fondamentale nella fase di analisi effettuata dalla Logica per riconoscere la presenza di problematiche all'interno della casa.

Attributi

- **dataOra:** Identifica il giorno e l'ora in cui è stata eseguita la rilevazione[1]
- **valore:** Valore raccolto dal Dispositivo durante la rilevazione[2]

4.2.7 Azione

Questa classe concettuale rappresenta le azioni eseguite da un Dispositivo (Attuatore).
Questo elemento è fondamentale nella fase di esecuzione, da parte dei dispositivi fisici, delle azioni dettate dalla Logica *o richieste dall'utente*.

4.3 Associazioni

4.3.1 Utente *comunicaCon* Middleware

Questa associazione rappresenta lo scambio di messaggi ed informazioni tra un *Utente* e il *Sistema*.

Cardinalità

Utente (1, *) $\longleftrightarrow$ Middleware (1, 1):
Uno o più utenti della casa comunicano col *Middleware* del sistema (rappresentabile anche come un elemento fisico, per esempio una centralina).

4.3.2 Middleware *comunicaCon* Logica

Questa associazione modella l'articolato processo di comunicazione tra *Middleware* e *Logica*.
Il *Middleware* invia le informazioni che riceve dai *Dispositivi* o dagli *Utenti* mentre la *Logica* comunica le *Azioni* che ha elaborato.

Cardinalità

Middleware (1, 1) $\longleftrightarrow$ Logica (1, 1):
All'interno del sistema SmartHome consideriamo i concetti di *Middleware* e *Logica* come elementi unici dell'archiettura.

[1] Nella modellazione del diagramma delle classi a livello di dominio, utilizzando lo strumento VisualParadigm, non è stato possibile associare il tipo di dato *DateTime* a questo attributo in quanto non presente nell'ambiente di realizzazione del modello.

[2] Per questo attributo è stato volutamente omesso il tipo di dato in quanto, nei diversi scenari di rilevazione, questo può essere di natura diversa. Per esempio nel caso di un sensore di temperatura e un sensore di movimento.

4.3.3 Middleware *comunicaCon* Dispositivo

Questa associazione rappresenta il processo di comunicazione bidirezionale tra *Middleware* e *Dispositivi*.
I *Dispositivi* inviano delle *Rilevazioni* o il loro stato corrente mentre il *Middleware* comunica le azioni da eseguire che ha ricevuto dalla *Logica*.

Cardinalità

Middleware (1, 1) $\longleftrightarrow$ Dispositivo (1, *):
Uno o più *Dispositivi* della casa interagiscono col *Middleware* del sistema.

4.3.4 Dispositivo *effettua* Rilevazione

Questa associazione rappresenta il processo con cui alcuni tipi di *Dispositivi* (Sensori) effettuano delle *Rilevazioni* all'interno della casa.

Cardinalità

Dispositivo (1, *) $\longrightarrow$ Rilevazione (0, *):
Uno o più *Dispositivi* della casa possono effettuare o meno delle *Rilevazioni*.
In questa associazione, dal lato della classe concettuale *Rilevazione*, abbiamo inserito cardinalità minima 0 perché la classe *Dispositivi* rappresenta anche gli attuatori che non possono effettuare nessuna *Rilevazione*.

4.3.5 Dispositivo *esegue* Azione

Questa associazione rappresenta il processo con cui alcuni tipi di *Dispositivi* (Attuatori) eseguono delle *Azioni* all'interno della casa.

Cardinalità

Dispositivo (1, *) $\longrightarrow$ Azione (0, *):
Uno o più *Dispositivi* della casa possono eseguire o meno delle *Azioni*.
In questa associazione, dal lato della classe concettuale *Azione*, abbiamo inserito cardinalità minima 0 perché la classe *Dispositivi* rappresenta anche i sensori che non possono eseguire nessuna *Azione*.

4.3.6 Dispositivo *appartieneA* Stanza

Con questa associazione a ciascun *Dispositivo* viene associata la *Stanza* in cui si trova all'interno della casa.

Cardinalità

Dispositivo (1, *) $\longrightarrow$ Stanza (1, 1):
Uno o più dispositivi appartengono a una stanza

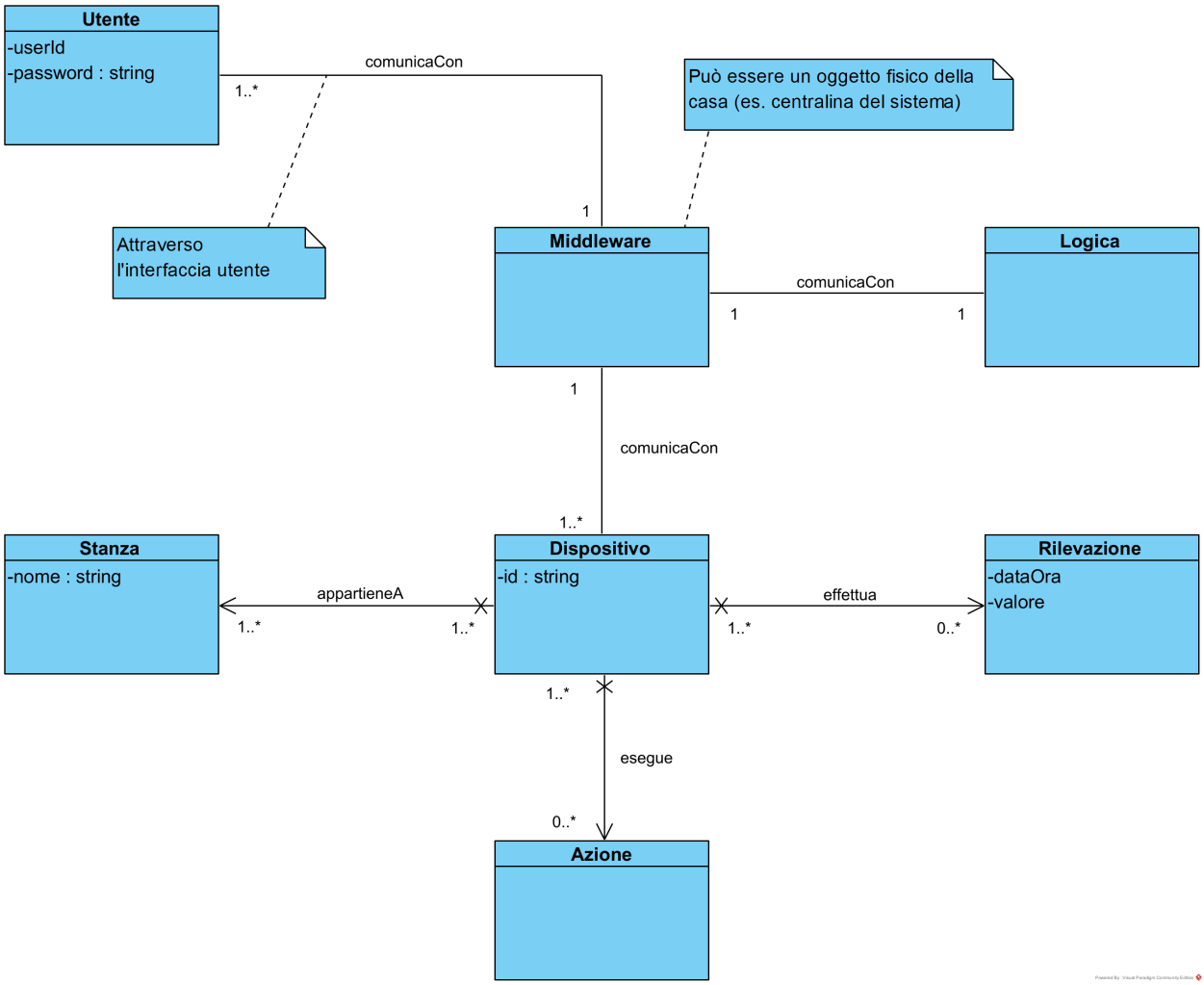

Immagine 4.1: Diagramma delle classi a livello di dominio

Capitolo 5

Diagrammi di Sequenza di Sistema

5.1 Introduzione

I diagrammi di sequenza di sistema sono uno strumento importante della fase di progettazione perché permettono di definire in maniera dettagliata la sequenza di operazioni necessarie alla realizzazione dei diversi scenari dei casi d'uso.
In questo capitolo riportiamo gli SSD che rappresentano gli scenari più importanti dei principali casi d'uso del sistema.

5.2 *Azioni richieste dall'utente*

Questo SSD rappresenta la maggior parte degli scenari che estendono il caso d'uso base ControlloManuale, ovvero tutte le situazioni in cui l'utente richiede al sistema di eseguire un'azione.
In particolare in questo SSD viene rappresentato il caso in cui l'utente richiede al sistema di accedere una luce della casa.

Messaggi del diagramma

- **richiediAccendiLuce(id):** Questo messaggio può essere considerato l'evento di questo SSD con il quale viene richiesta l'operazione di sistema per accendere la luce del dispositivo di cui si passa l'identificativo come parametro del messaggio.
- **verificaAzione:** Con questa operazione il sistema verifica che l'azione richiesta dall'utente non crei conflitti con altre operazioni in esecuzione.
- **accendiLuce:** Questa operazione viene eseguita solo se la verifica precedente dà esito positivo.
  Con questo messaggio il sistema invia al dispositivo indicato dall'utente la richiesta di accendere la luce.

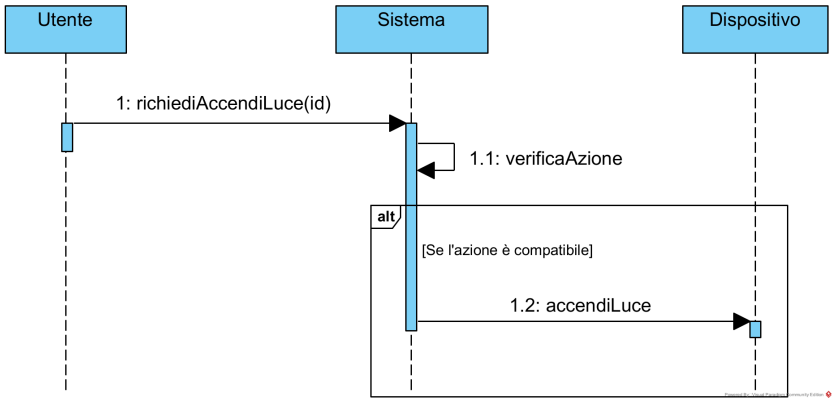

Immagine 5.1: SSD per il caso d'uso AccendiLuceUtente

5.3 Azioni richieste dal sistema

5.3.1 Caso d'uso SpegniLuceSistema

Questo SSD rappresenta la maggior parte degli scenari che estendono il caso d'uso base *ControlloAutomatico*, ovvero tutte le situazioni in cui il sistema richiede ai dispositivi fisici (attuatori) di eseguire un'azione.
In particolare in questo SSD, simile nei messaggi a quello mostrato al punto precedente, viene rappresentato il caso in cui il sistema richiede ad un dispositivo di spegnere la luce.

Messaggi del diagramma

- **richiediSpegniLuce(id):** Questo messaggio può essere considerato l'evento di questo SSD con il quale la logica richiede l'operazione di sistema per spegnere la luce del dispositivo di cui si passa l'identificativo come parametro del messaggio.
- **verificaAzione:** Con questa operazione il sistema verifica che l'azione richiesta dalla logica non crei conflitti con altri scenari in esecuzione.
- **spegniLuce:** Questa operazione viene eseguita solo se la verifica precedente dà esito positivo.
  Con questo messaggio il sistema invia al dispositivo indicato nel messaggio iniziale la richiesta di spegnere la luce.

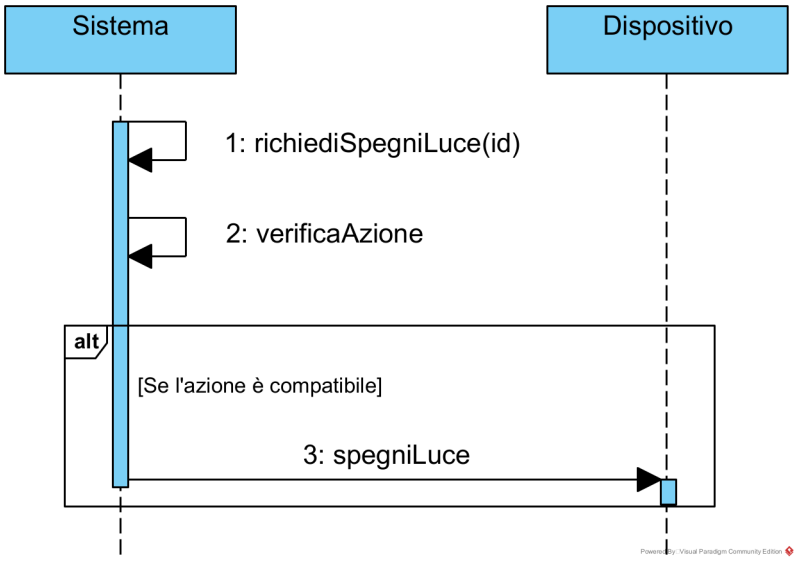

Immagine 5.2: SSD per il caso d'uso SpegniLuceSistema

5.3.2 Caso d'uso ProceduraEmergenzaIntrusione

In questo SSD rappresentiamo un caso particolare degli scenari che estendono il caso d'uso base *ControlloAutomatico*, quello in cui il sistema rileva la presenza di intrusi ed avvia una procedura di emergenza.

Messaggi del diagramma

- **segnalaIntrusione:** Questo messaggio può essere considerato l'evento di questo SSD con il quale il dispositivo fisico (sensore) segnala al sistema la presenza di un intruso all'interno della casa.
- **verificaAzione:** Con questa operazione il sistema verifica che l'azione richiesta dalla logica non crei conflitti con altri scenari in esecuzione.
  Se questa verifica dà esito positivo, il sistema procederà ad eseguire le operazioni successive.
- **accendiLuciCasa:** Con questo messaggio il sistema richiede a tutte le luci dell'abitazione di accendersi.

- **accendiSirena:** Con questo messaggio il sistema richiede alla sirena del sistema di allarme di accendersi.
- **notificaUtente:** Con questo messaggio il sistema invia una notifica all'utente avvisandolo della rilevazione di un'intrusione all'interno dell'abitazione.

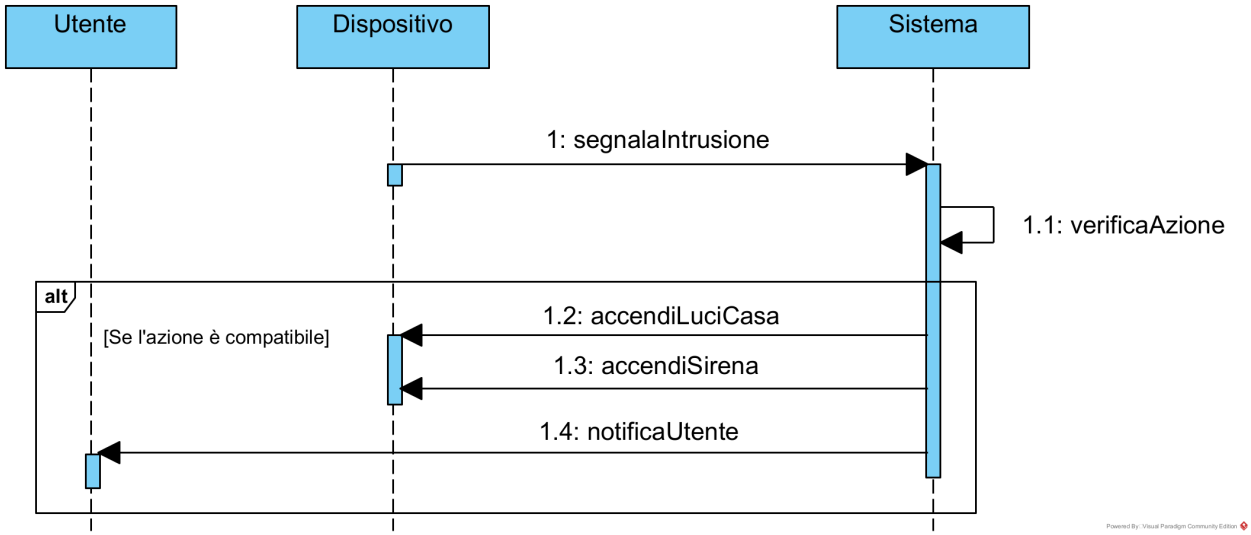

Immagine 5.3: SSD per il caso d'uso ProceduraEmergenzaIntrusione

Capitolo 6

Diagrammi di Sequenza di Progettazione

6.1 Introduzione

I diagrammi di sequenza di progettazione sono un passo ulteriore dalla fase di modellazione, legata ai concetti del dominio, verso la fase di progettazione, più vicina agli oggetti software.
Questi diagrammi sono uno strumento importante attraverso il quale si riescono a definire in maniera più concreta i metodi che verranno poi implementati nelle classi software.
In questo capitolo, come in quello precedente, riportiamo i diagrammi di sequenza di progettazione che rappresentano gli scenari più importanti dei principali casi d'uso del sistema.

Nei diagrammi mostrati in questo capitolo i messaggi e le operazioni sono fortemente legati alla struttura dell'architettura e del framework utilizzato.
Inoltre alcune delle classi utilizzate i questo capitolo sono proprie della fase di progettazione del diagramma delle classi.

6.2 *Azioni richieste dall'utente*

Questo diagramma rappresenta la maggior parte degli scenari in cui l'utente richiede al sistema di far eseguire un'azione ad uno specifico attuatore.
In particolare in questo diagramma di sequenza viene rappresentato il caso in cui l'utente richiede al sistema di accedere una luce della casa.

Messaggi del diagramma

- **useDevice(id):** Attraverso questo messaggio l'utente richiede l'utilizzo di un Attuatore specificandone l'id.
- **checkConflicts():** Con questa operazione il sistema verifica che l'azione richiesta dall'utente non crei conflitti con altri scenari in esecuzione.
- **changeState(label):** Questa operazione viene eseguita solo se la verifica precedente dà esito positivo. Con questo messaggio il sistema invia al dispositivo indicato dall'utente la richiesta avviare la transizione di stato.

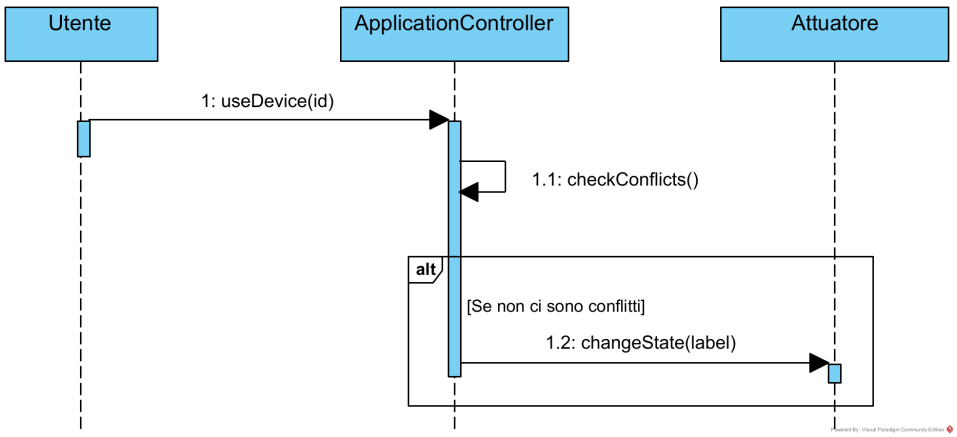

Immagine 6.1: Diagrammi di Sequenza di Progettazione per il caso d'uso AccendiLuceUtente

6.3 Azioni richieste dal sistema

Questo diagramma rappresenta la maggior parte degli scenari in cui il sistema richiede ad uno specifico attuatore di eseguire un'azione.
In particolare in questo diagramma di sequenza viene rappresentato il caso in cui il sistema richiede ad un dispositivo di spegnere la luce.

Messaggi del diagramma

- **useDevice(id):** Attraverso questo messaggio l'Agente che si occupa del sistema di illuminazione richiede l'utilizzo di un Attuatore specificandone l'id.
- **checkConflicts():** Con questa operazione il sistema verifica che l'azione richiesta dall'utente non crei conflitti con altri scenari in esecuzione.
- **changeState(label):** Questa operazione viene eseguita solo se la verifica precedente dà esito positivo. Con questo messaggio il sistema invia al dispositivo indicato dall'utente la richiesta avviare la transizione di stato.

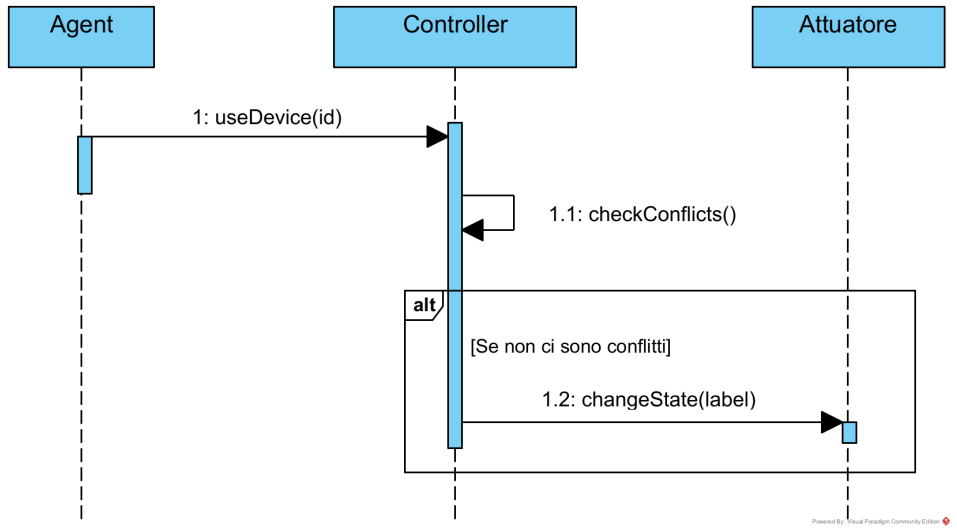

Immagine 6.2: Diagrammi di Sequenza di Progettazione per il caso d'uso SpegniLuceSistema

Capitolo 7

Diagramma delle classi

7.1 Introduzione

In questa fase della progettazione vogliamo realizzare con oggetti software, attraverso classi, interfacce e le relative associazione, gli elementi che nel modello di dominio sono stati rappresentati dal punto di vista concettuale.
Inoltre, durante la modellazione del diagramma delle classi, abbiamo applicato diversi design pattern per l'assegnazione delle responsabilità agli oggetti software.

7.2 Classificatori

7.2.1 Device

Questo oggetto software rappresenta quella classe che, in fase di modellazione all'interno del Modello di Dominio, abbiamo chiamato Dispositivo.
A differenza della classe concettuale, dove l'oggetto Dispositivo rappresentava tutte le possibili tipologie di device presenti in una casa, nel diagramma delle classi abbiamo separato il concetto di dispositivo in tre oggetti software distinti. In questo modo si ha la possibilità di associare le corrette funzionalità a seconda del tipo di device che la classe rappresenta.
La classe Device generalizza le diverse tipologie di dispositivi, Sensore e Attuatore, includendo attributi ed operazioni comuni.

Attributi

id:Long	Identificatore auto-generato per la rappresentazione univoca del dispositivo nel DB
label:String	Nome utilizzato per identificare il dispositivo negli endpoint delle API
category:Category	Attributo di tipo Enumeration che associa ogni dispositivo una tipologia di device tra quelle gestite dal sistema.
room:Room	Attributo che rappresenta la stanza della casa in cui si trova il dispositivo.
deviceType:Boolean	Attributo che permette di distinguere se l'oggetto è un sensore o un attuatore.
rilevations:Set<Rilevation>	Insieme delle rilevazioni effettuate dal dispositivo.

Operazioni

La classe contiene, oltre ai costruttori, i metodi *Getter* e *Setter* per i suoi attributi

Design Pattern

Remote Proxy	Tutti gli oggetti di tipo Device fanno riferimento, attraverso l'attributo label, a degli oggetti remoti creati in una classe del framework PatrIot dove ne viene definito il comportamento e, per i sensori, la tipologia di dati che possono generare.

7.2.2 Sensor

Questa classe software estende la classe Device e rappresenta la categoria di dispositivi Sensore.

Operazioni

Questa classe, oltre ai costruttori e alle operazioni ereditate dalla classe Device, definisce il seguente metodo:

getDataFeed():double Questo metodo permette, attraverso l'utilizzo del protocollo di comunicazione CoAP, di richiedere al sensore di effettuare una rilevazione e di restituirne il valore.

7.2.3 Actuator

Questa classe software estende la classe Device e rappresenta la categoria di dispositivi Attuatore.

Operazioni

Questa classe, oltre ai costruttori e alle operazioni ereditate dalla classe Device, definisce i seguenti metodi:

getCurrentState():String Questo metodo permette, attraverso l'utilizzo del protocollo di comunicazione CoAP, di ottenere lo stato dell'attuatore.

controlSignal():void Questo metodo permette, attraverso l'utilizzo del protocollo di comunicazione CoAP, di richiedere all'attuatore di effettuare una transizione di stato.

7.2.4 Rilevation

Questa classe astrae con un oggetto software il concetto di rilevazione effettuata da un dispositivo di tipo *Sensor*.

Attributi

id:Long Identificatore auto-generato per la rappresentazione univoca della rilevazione nel database

value:Double Valore effettivo della rilevazione

valueType:String Tipo di dato dell'attributo *value*[1]

timestamp:Timestamp Data e ora in cui è stata effettuata la rilevazione

device:Device Dispositivo che ha effettuato la rilevazione

Operazioni

La classe contiene, oltre ai suoi costruttori, i metodi *Getter* per i suoi attributi.
Uno dei costruttori accetta come parametro una stringa in formato JSON dalla quale estrae i dati che utilizza per inizializzare l'oggetto.

7.2.5 rilevationController

Questa classe si occupano di gestire le richieste di operazioni con la base di dati, per oggetti di tipo *Rilevation*, ricevute dall'interfaccia utente.

Attributi

rilevationService:RilevationService Attraverso questo oggetto la classe recupera dal database informazioni sugli oggetti di tipo *Rilevation*

[1]Questo attributo è stato inserito per poter distinguere i valori di tipo double, generati ad esempio da un sensore di temperatura, da quelli di tipo boolean che possono essere generati da un sensore di movimento.

Operazioni

Oltre al costruttore la classe definisce il seguente metodo:

getLastTemperature() Attraverso questo metodo è possibile ottenere, recuperandolo dal DB, l'ultimo oggetto di tipo *Rilevation* in cui è stato registrata una temperatura

Design Pattern

Controller Questa classe raccoglie i messaggi inviati dall'interfaccia utente e individua quale operazione di sistema eseguire

7.2.6 Room

Questo oggetto software rappresenta quella classe che, in fase di modellazione all'interno del Modello di Dominio, abbiamo chiamato Stanza.
Attraverso questa classe abbiamo la possibilità di identificare tutti i *Device* che appartengono alla stessa stanza; permettendo quindi una più facile associazione tra problema ed attuatore con il quale risolverlo.

Attributi

id:Long Identificatore auto-generato per la rappresentazione univoca della stanza nel database

name:String Nome utilizzato per identificare il dispositivo negli endpoint delle API

Operazioni

La classe, oltre ai costruttori, contiene i metodi *Getter* e *Setter* per i suoi attributi

7.2.7 deviceController

Questa classe si occupano di gestire le richieste di operazioni con la base di dati, per oggetti di tipo Device, ricevute dall'interfaccia utente.

Attributi

deviceService:DeviceService Attraverso questo oggetto la classe recupera dal database informazioni sugli oggetti di tipo *Device*

Operazioni

Oltre al costruttore la classe definisce i seguenti metodi:

getDeviceValue(Long deviceId) Attraverso questo metodo è possibile ottenere, recuperandolo dal DB, l'oggetto di tipo *Actuator* o *Sensor* a cui corrisponde l'id passato come parametro

getDeviceValue(Long deviceId) Attraverso questo metodo è possibile eliminare dal database il dispositivo a cui corrisponde l'id passato come parametro

Design Pattern

Controller Questa classe raccoglie i messaggi inviati dall'interfaccia utente e individua quale operazione di sistema eseguire

7.2.8 deviceService

Questa classe fa da intermediario tra il sistema e il database offrendo dei metodi per il recupero di oggetti di tipo *Device* dalla base di dati.

Attributi

deviceRepository:DeviceRepository Questa interfaccia associa ai metodi utilizzati in questa classe le query utilizzate per recuperare dati dal database

Operazioni

Oltre al costruttore la classe definisce i seguenti metodi:

getDeviceByRoom(Room room):List<Device>	Attraverso questo metodo è possibile ottenere, recuperandola dal database, una lista contenente tutti i dispositivi appartenenti alla stanza passata come parametro
getDeviceByLabel(String label):Device	Attraverso questo metodo è possibile ottenere, recuperandolo dal database, l'oggetto di tipo Device a cui corrisponde la label passata come parametro
getDeviceById(Long deviceId):Device	Attraverso questo metodo è possibile ottenere, recuperandolo dal database, l'oggetto di tipo Device a cui corrisponde l'id passato come parametro
getAllDevices():List<Device>	Attraverso questo metodo è possibile ottenere, recuperandola dal database, una lista contenente tutti i dispositivi del sistema.
deleteDevice(Long deviceId):void	Attraverso questo metodo è possibile eliminare dal database l'oggetto di tipo Device a cui corrisponde l'id passato come parametro

Design Pattern

Pure Fabrication	Questa classe artificiale è utile per sostenere coesione alta, accoppiamento basso e riuso
Indirection	Questa classe permette di evitare un accoppiamento diretto tra le classi che contengono la logica applicativa e le interfacce che comunicano con il database. Questo garantisce un basso accoppiamento e un alto riuso.

7.2.9 rilevationService

Questa classe fa da intermediario tra il sistema e il database offrendo dei metodi per il salvataggio ed il recupero di oggetti di tipo *Rilevation* dalla base di dati.

Attributi

rilevationRepository:RilevationRepository	Questa interfaccia associa ai metodi utilizzati in questa classe le query utilizzate per salvare o recuperare dati dal database

Operazioni

La classe, oltre al costruttore, implementa i seguenti metodi:

getRilevationByDevice(Device device):List<Rilevation>	Attraverso questo metodo è possibile ottenere, recuperandola dal database, una lista contenente tutte le rilevazioni effettuate dall'oggetto di tipo *Device* passato come parametro
getRilevationById(Long id):Rilevation	Attraverso questo metodo è possibile ottenere, recuperandolo dal database, la rilevazione a cui corrisponde l'identificatore passato come parametro
saveRilevation(Rilevation rilevation):void	Attraverso questo metodo è possibile salvare sul database la rilevazione passata come parametro

Design Pattern

Pure Fabrication Questa classe artificiale è utile per sostenere coesione alta, accoppiamento basso e riuso

Indirection Questa classe permette di evitare un accoppiamento diretto tra le classi che contengono la logica applicativa e le interfacce che comunicano con il database.
Questo garantisce un basso accoppiamento e un alto riuso.

7.2.10 *Controller*

Questa classe astratta generalizza le entità del sistema che si occupano di gestire i messaggi ricevuti dai dispositivi simulati dal framework PatrIot.

Attributi

deviceService:DeviceService Attraverso questo oggetto la classe recupera dal database informazioni sugli oggetti di tipo *Device*

rilevationService:RilevationService Attraverso questo oggetto la classe salva sul database i nuovi oggetti di tipo *Rilevazione*

Operazioni

La classe, oltre al costruttore, definisce il seguente metodo astratto:

receiveSensorData(jsonData:JSONObject):void Attraverso questo metodo le classi figlie gestiranno, a seconda della loro area di interesse, le rilevazioni ricevute dai sensori attraverso chiamate al loro endpoint.
I dati delle rilevazioni sono contenuti in una stringa in formato JSON passata come parametro.

Design Pattern

Polymorphism Questa classe prevede l'implementazione del metodo astratto *receiveSensorData* per le classi che la estendono. Ciascuna di esse avrà un'implementazione differente a seconda dell'area di interesse che gestisce

Creator Le classi che estendono *Controller* sono responsabili della creazione degli oggetti di tipo Rilevazione, *Agente* e in alcuni casi Emergenza

Information Expert Questa classe possiede tutte le informazioni necessarie per scegliere quale tipo di agente creare tra quelli che estendono la classe astratta *Agente* e per creare gli oggetti di tipo Rilevazione ed Emergenza.

Controller Le classi che estendono *Controller* hanno il compito raccogliere i messaggi inviati dai sensori, formalizzarne le informazioni per poi farle analizzare alla logica del sistema.

Classi Figlie

Le classi che estendono *Controller* implementano in maniera differente il metodo *receiveSensorData*, adattandolo alle caratteristiche dell'area di interesse di cui si occupano.

- **GasController:** Gestisce le rilevazioni effettuate dai sensori per le perdite di gas
- **MovementController:** Gestisce le rilevazioni effettuate dai sensori di movimento
- **SmokeController:** Gestisce le rilevazioni effettuate dai sensori di fumo
- **ThermometerController:** Gestisce le rilevazioni effettuate dai sensori di temperatura
- **AlarmController:** Gestisce i messaggi per attivare o disattivare il sistema di allarme

I primi tre tipi di controller dell'elenco hanno anche l'attributo emergenzaRepo:emergenzaRepository attraverso il quale si possono registrare sul database situazioni di emergenza legate a intrusioni, fughe di gas o presenza di fumo all'interno della casa.

7.2.11 Emergenza

Questa classe astrae il concetto di situazione di emergenza che si verifica all'interno della casa.
Nel nostro sistema possono verificarsi tre tipi di situazioni di emergenza:

- Fuga di gas
- Intrusione
- Presenza di fumo

Attributi

id:Long	Identificatore auto-generato per la rappresentazione univoca dell'oggetto nel DB
timestamp:Timestamp	Data e ora a cui si è verificata l'emergenza
code:EmergencyCode	Attributo di tipo Enumeration che associa ogni oggetto il tipo di emergenza che rappresenta.
room:Room	Attributo che rappresenta la stanza della casa in cui si è verificata l'emergenza
isEmergencyRead:Boolean	Attributo con il quale si verifica che l'utente abbia preso visione della presenza di una situazione di emergenza

Operazioni

La classe contiene, oltre ai costruttori, i metodi *Getter* per i suoi attributi

7.2.12 EmergenzaService

Questa classe fa da intermediario tra il sistema e il database offrendo dei metodi per il salvataggio ed il recupero di oggetti di tipo *Emergenza* dalla base di dati.

Attributi

emergenzaRepository:EmergenzaRepository	Questa interfaccia associa ai metodi utilizzati in questa classe le query utilizzate per recuperare dati dal DB

Operazioni

getEmergenzaByRoom(Room room):List<Emergenza>	Attraverso questo metodo è possibile ottenere, recuperandola dal database, una lista contenente tutte le emergenze verificatesi nella stanza passata come parametro
getAllEmergenze():List<Emergenza>	Attraverso questo metodo è possibile recuperare dal database una lista contenente tutte le emergenze verificatesi nel sistema
getPendingEmergenze():List<Emergenza>	Attraverso questo metodo è possibile recuperare dal database, una lista contenente tutte le emergenze verificatesi nel sistema di cui l'utente non ha ancora preso visione
getEmergenzaById(Long id):Emergenza	Attraverso questo metodo è possibile recuperare dal DB la situazione di emergenza a cui corrisponde l'id passato come parametro
saveEmergenza(Emergenza emergenza):void	Attraverso questo metodo è possibile salvare nel DB l'oggetto di tipo *Emergenza* passata come parametro

updateEmergenzaStatus(Long emergenzaId, boolean newStatus):void
Attraverso questo metodo è possibile aggiornare il valore dell'attributo isEmergencyRead per l'oggetto a cui corrisponde l'id passato come parametro

Design Pattern

Pure Fabrication Questa classe artificiale è utile per sostenere coesione alta, accoppiamento basso e riuso

Indirection Questa classe permette di evitare un accoppiamento diretto tra le classi che contengono la logica applicativa e le interfacce che comunicano con il database.
Questo garantisce un basso accoppiamento e un alto riuso.

7.2.13 EmergenzaController

Questa classe si occupano di gestire le richieste di operazioni con la base di dati ricevute dall'interfaccia utente.

Attributi

emergenzaService:EmergenzaService Attraverso questo oggetto la classe ottiene dal database delle informazioni per la creazione degli oggetti di tipo *Emergenza*

Operazioni

Oltre al costruttore la classe implementa i seguenti metodi:

getAllEmergenze():List<Emergenza> Con questo metodo è possibile ottenere, usando l'oggetto emergenzeService, una lista contenente tutte le emergenza verificatesi nel sistema

getPendingEmergenze():List<Emergenza> Con questo metodo è possibile ottenere, usando l'oggetto emergenzeService, una lista contenente tutte le emergenza verificatesi nel sistema di cui l'utente non ha ancora preso visione

updateEmergenza(Long emergenzaId):void Con questo metodo è possibile impostare a true il valore dell'attributo isEmergencyRead per l'oggetto a cui corrisponde l'id passato come parametro

Design Pattern

Controller Questa classe raccoglie i messaggi inviati dall'interfaccia utente e individua l'operazione di sistema da eseguire.

7.2.14 *Agente*

Questa classe astratta generalizza le entità del sistema che si occupano di analizzare le rilevazioni effettuate dai sensori, riconoscere la presenza di scenari[2] che richiedono l'intervento del sistema ed individuare le azioni necessarie a risolverle.

Attributi

deviceService:DeviceService Questo oggetto viene utilizzato nelle diverse strategie di azione per recuperare dal database informazioni sui dispositivi del sistema

rilevazione:Rilevation Rilevazione che l'agente deve analizzare

Operazioni

La classe, oltre al costruttore, definisce il seguente metodo astratto:

run:void In questo metodo, le classi che estendono *Agente*, scelgono ed eseguono la strategia di azione a seconda dell'area di interesse che gestiscono e della rilevazione che analizzano

[2]Si intendono quegli scenari in cui le rilevazioni effettuate dai sensori evidenziano la violazione delle preferenza impostate dell'utente o di alcune regole base del sistema

Design Pattern

Polymorphism	Questa classe astratta prevede l'implementazione del metodo *run* per le classi che la estendono. Ciascuna di esse avrà un'implementazione differente a seconda dell'area di interesse che gestisce
Creator	Le classi che estendono *Agente* creano gli oggetti che rappresentano la strategia di azione impiegata dall'agente
Information Expert	Le classi che estendono *Agente* creano gli oggetti strategia perché conoscono l'area di interesse che gestiscono e capiscono, attraverso la rilevazione che ricevono, se è necessario eseguire delle azioni.
Strategy	Le classi che estendono *Agente* scelgono tra diverse strategie di azione a seconda dell'area che gestiscono e di eventuali problematiche che individuano analizzando le rilevazioni che ricevono

Classi Figlie

- **AgenteAllarme:** Entità che si occupa di riconoscere e gestire le intrusioni nell'abitazioni
- **AgenteLuce:** Entità che si occupa di gestire le luci dell'abitazioni
- **AgenteTemperatura:** Entità che si occupa di gestire i dispositivi per il controllo della temperatura dell'abitazione
- **AgentePulizia:** Entità che si occupa di gestire i dispositivi per la pulizia dell'abitazione
- **AgentePericoli:** Entità che si occupa di riconoscere e gestire le situazioni di pericolo nell'abitazioni (fughe di gas e presenza di fumo)

7.2.15 AgenteStatus

Questa classe rappresenta lo stato degli agenti per le diverse aree di interesse gestite dal sistema.

Attributi

Gli attributi di questa classe rappresentano lo stato degli agenti per le diverse aree di interesse gestite dal sistema. Se l'attributo vale true significa che possono essere creati agenti di quella categoria e di conseguenza si possono eseguire azioni per quella determinata area di interesse.

- allarme:AtomicBoolean
- luci:AtomicBoolean
- pericoli:AtomicBoolean
- pulizia:AtomicBoolean
- temperatura:AtomicBoolean

Operazioni

La classe, oltre al costruttore, implementa i metodi *Getter* e *Setter* per i suoi attributi

Design Pattern

Pure Fabrication Questa classe artificiale è utile per sostenere coesione alta, accoppiamento basso e riuso

7.2.16 Strategy

Questa interfaccia, propria del design pattern Strategy, rappresenta le diverse strategie di azione nella risoluzione delle problematiche di una casa. Le classi che implementano questa interfaccia si differenziano per area di interesse e tipologie di scenari che gestiscono.

Operazioni

execute(Rilevation rilevazione, DeviceService deviceService):void
Nelle classi che implementano questa interfaccia, il metodo *execute* contiene la logica applicativa del sistema, ovvero le azioni scelte per risolvere le problematiche identificate dagli agenti.

Design Pattern

Polymorphism Questa interfaccia prevede l'implementazione del metodo astratto *execute* per le classi che la implementano. Ciascuna di esse avrà una codifica differente del metodo a seconda dell'area di interesse e dei tipi di scenari che gestisce.

Classi che implementano l'interfaccia

- StrategyAllarme
- StrategyGas
- StrategyLuceGiorno
- StrategySmoke
- StrategyTemperaturaEstate
- StrategyTemperaturaInverno

7.2.17 roomService

Questa classe fa da intermediario tra il sistema e il database offrendo dei metodi per l'aggiunta, l'eliminazione o il recupero di oggetti di tipo *Room* dalla base di dati.

Attributi

roomRepository:RoomRepository Questa interfaccia associa ai metodi utilizzati in questa classe le query utilizzate per recuperare dati dal database

Operazioni

Oltre al costruttore la classe implementa i seguenti metodi:

getRooms():List<Room> Attraverso questo metodo è possibile ottenere, recuperandola dal DB, una lista contenente tutte le stanze del sistema

addNewRoom(Room room):void Attraverso questo metodo è possibile aggiungere al database la stanza passata come parametro

deleteRoom(Long roomId):void Attraverso questo metodo è possibile eliminare dal database la stanza a cui corrisponde l'id passato come parametro

Design Pattern

Pure Fabrication Questa classe artificiale è utile per sostenere coesione alta, accoppiamento basso e riuso

7.2.18 RoomController

Questa classe si occupano di gestire le richieste di operazioni con la base di dati ricevute dall'interfaccia utente.

Attributi

roomService:RoomService Attraverso questo oggetto la classe ottiene dal database delle informazioni per la creazione, l'eliminazione o il recupero degli oggetti di tipo *Room*

Operazioni

Questa classe, oltre al construttore, implementa i seguenti metodi:

getAllRooms() Con questo metodo è possibile ottenere, attraverso l'oggetto roomService, una lista contenente tutte le stanze del sistema

registerNewRoom(Room room) Con questo metodo è possibile aggiungere al database, usando l'oggetto roomService, la stanza passata come parametro

deleteRoom(Long roomId) Con questo metodo è possibile eliminare dal database, usando l'oggetto roomService, la stanza a cui corrisponde l'id passato come parametro

Design Pattern

Controller Questa classe raccoglie i messaggi inviati dall'interfaccia utente e individua quale operazione di sistema eseguire.

7.2.19 User

Attributi

id:Long Identificatore auto-generato per la rappresentazione univoca dell'utente nel database

name:String Nome dell'utente

email:String Email dell'utente

Operazioni

La classe, oltre al costruttore, contiene i metodi *Getter* e *Setter* per i suoi attributi

7.2.20 userService

Questa classe fa da intermediario tra il sistema e il database offrendo dei metodi per l'aggiunta, l'eliminazione o il recupero di oggetti di tipo *User* dalla base di dati.

Attributi

userRepository:UserRepository Questa interfaccia associa ai metodi utilizzati in questa classe le query utilizzate per recuperare dati dal database

Operazioni

La classe, oltre al costruttore, implementa i seguenti metodi:

getUsers():List<User> Attraverso questo metodo è possibile recuperare dal database una lista contenente tutti gli utenti del sistema

addNewUser(User user):void Attraverso questo metodo è possibile salvare nel database un nuovo utente del sistema

deleteUser(Long userId):void Attraverso questo metodo è possibile eliminare dal database l'utente del sistema a cui corrisponde l'id passato come parametro

updateUser(Long userId, String name, String email):void Attraverso questo metodo è possibile aggiornare nel database i dati dell'utente del sistema a cui corrisponde l'id passato come parametro

Design Pattern

Pure Fabrication Questa classe artificiale è utile per sostenere coesione alta, accoppiamento basso e riuso

7.2.21 UserController

Questa classe si occupano di gestire le richieste di operazioni con la base di dati ricevute dall'interfaccia utente.

Attributi

userService:UserService Attraverso questo oggetto la classe ottiene dal database delle informazioni per la creazione degli oggetti di tipo *User*

Operazioni

Oltre al costruttore la classe definisce i seguenti metodi:

getUsers():List<User> Con questo metodo è possibile ottenere, attraverso l'oggetto userService, una lista contenente tutti gli utenti del sistema

registerNewUser(User user):void Con questo metodo è possibile salvare nel database, usando l'oggetto userService, un nuovo utente del sistema

deleteUser(Long userId):void Con questo metodo è possibile eliminare dal database, utilizzando l'oggetto userService, l'utente del sistema a cui corrisponde l'id passato come parametro

updateUser(Long userId, String name, String email):void Con questo metodo è possibile aggiornare nel database, usando l'oggetto userService, i dati dell'utente del sistema a cui corrisponde l'id passato come parametro

Design Pattern

Controller Questa classe raccoglie i messaggi inviati dall'interfaccia utente e individua l'operazione di sistema da eseguire.

7.2.22 SmartHomeApplication

In questa classe inizializziamo i dispositivi che compongono il sistema, legandoli ai loro corrispettivi, realizzati nel framework PatrIot, attraverso l'attributo label.

7.2.23 HomeSimulation

In questa classe del framework PatrIot definiamo i dispositivi del sistema, impostando i dati che possono generare ed il comportamento che possono assumere.

7.3 Pattern Architetturali

7.3.1 Table Data Gateway

Questo pattern viene rappresentato dalle coppie formate da classe *X*Service e interfaccia *X*Repository.
Queste coppie, che si differenziano per la tabella del database che gestiscono, sono le entità dell'applicazione che permettono l'interazione con il DB.

7.3.2 Data Mapper

Anche questo pattern, molto simile al Table Data Gateway, viene rappresentato dalle coppie formate da classe *X*Service e interfaccia *X*Repository. Ciascuna coppia rappresenta un mapper o DAO, ovvero delle entità che permettono un trasferimento bidirezionale dei dati tra DB e sistema, mantenendoli isolati l'uno dall'altro.

7.3.3 Identity Field Pattern

Questo pattern architetturale è direttamente collegato al Table Data Gateway in quanto abbiamo assegnato ad ogni classe che viene rappresentata come tabella nel database, un attributo aggiuntivo di tipo Long. Questo attributo è un identificatore auto-generato che permette di rendere univoco nella propria tabella, ciascun oggetto del sistema.

Un pattern architetturale non utilizzato

Il sistema che abbiamo realizzato basa gran parte della comunicazione tra i suoi componenti su chiamate ad API. Una sezione importante dell'applicazione che utilizza questo sistema per comunicare è l'invio delle rilevazioni da parte dei sensori.
Abbiamo deciso di separare la gestione delle chiamate contenenti le rilevazioni in diversi Controller, separati per tipologia di sensore di cui gestiscono le chiamate, effettuate ad endpoint dedicati.
Questa soluzione si contrappone all'utilizzo del pattern architetturale Application Controller con il quale tutte le chiamate da parte dei sensori, e la logica necessaria per gestirle, si sarebbero concentrate su un unico endpoint e in una sola classe.

7.4 Design Principles

7.4.1 SOLID

- **Single Responsibility Principle:** Abbiamo utilizzato questo principio nella realizzazione dei diversi Controller che gestiscono l'invio delle rilevazioni da parte dei sensori.
  Ciascuna classe che estende *Controller*, infatti, ha la sola responsabilità di gestire il tipo di rilevazione inviato dai sensori dell'area di interesse di cui si occupa.

- **Open Closed Principle:** Questo principio può essere riconosciuto nei casi in cui abbiamo definito delle operazioni polimorfe, realizzate da diverse classi che estendevano una classificatore astratto (es. *Controller*) oppure implementavano una interfaccia (es. Strategy).
  L'uso di questo pattern è fondamentale per mantenere il codice facile da mantenere e modificare.

- **Dependency Inversion Principle:** Abbiamo impiegato questo principio nella progettazione degli oggetti Controller, dove abbiamo definito il metodo astratto *getSensorData* che ogni classe "figlia" implementava a seconda dell'area di interesse che gestiva, e nell'utilizzo del design pattern Strategy, con cui tutte strategie di azione implementano in maniera diversa il metodo astratto *execute* in base al tipo di problemi che ogni strategia deve risolvere.
  L'uso di questo principio permette di avere codice riusabile e flessibile.

7.4.2 PHAME

- **Abstraction:** Ritroviamo questo principio nella realizzazione delle classi Device, Actuator e Sensor.
  Nella prima classe cerchiamo di generalizzare il più possibile le altre due, racchiudendo attributi ed operazioni comuni.
  L'implementazione delle classi Actuator e Sensor è necessaria alla sola definizione dei differenti comportamenti degli oggetti.

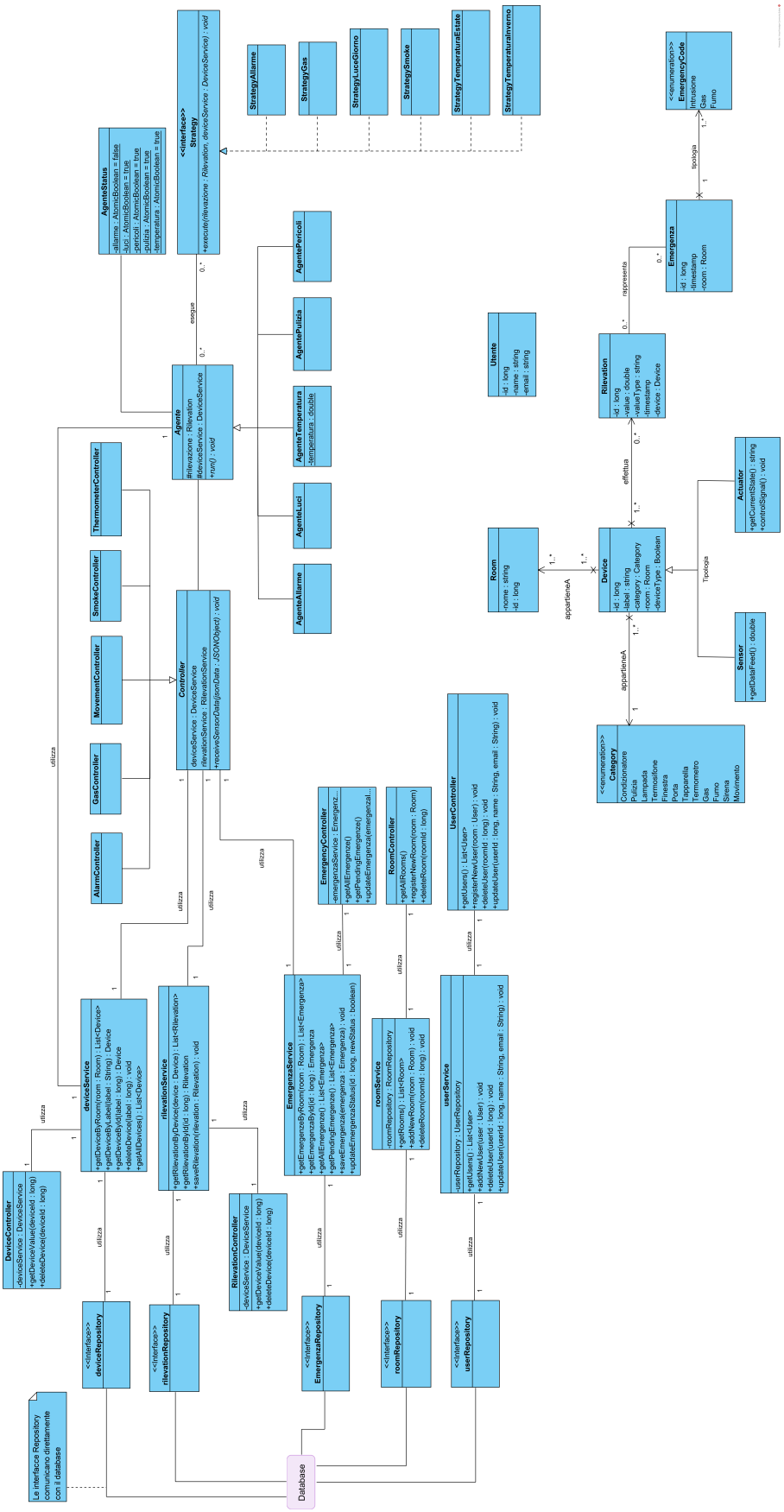

Immagine 7.1: Diagramma delle classi a livello di progettazione

Capitolo 8

Architettura Software

8.1 Introduzione

Dopo aver identificato le classi software che compongono il sistema le abbiamo organizzate in packages per costruire l'architettura software dell'applicazione.
Abbiamo utilizzato una struttura dell'architettura a strati stretta utilizzando i tre principali layer:

- **Interfaccia Utente:** Questo strato include tutti gli elementi del sistema che compongono l'interfaccia attraverso la quale l'utente riceve informazioni e comunica col sistema
- **Logica Applicativa:** Questo strato contiene le classi software che rappresentano gli elementi principali del dominio (dispositivi e stanze), quelle che contengono la logica del sistema (agenti) e quelle che fanno da tramite con lo strato superiore (controller)
- **Servizi tecnici:** Questo strato include il principale servizio che supporta il funzionamento del sistema, ovvero il framework PatrIot, unito al servizio per il supporto delle API, Spring, e a quello per la gestione del database, Hibernate.

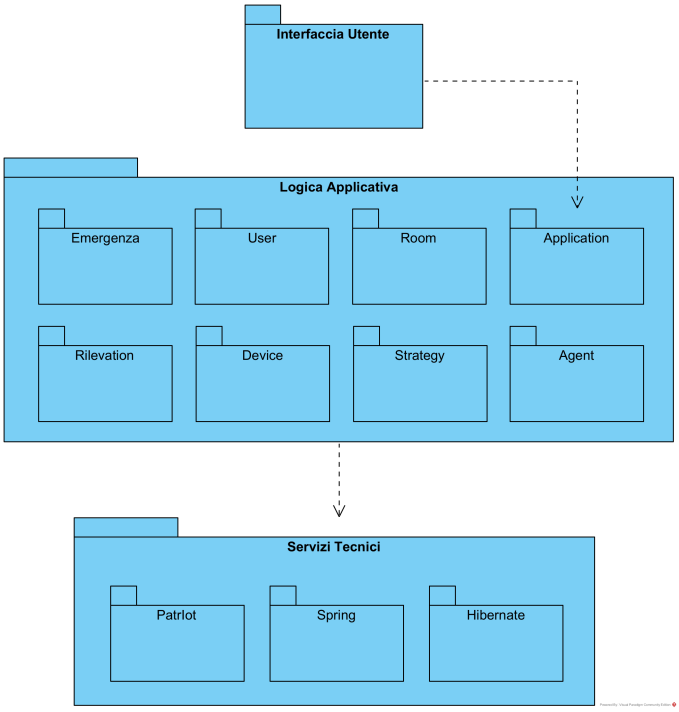

Immagine 8.1: Diagramma dell'Architettura Software

Capitolo 9

Diagrammi di Macchine a Stati

9.1 Introduzione

L'uso di questi diagrammi permette di avere un'ulteriore punto di vista sul comportamento dinamico del sistema e dei suoi componenti in termini di stati che vengono assunti e transizioni tra essi.
In questi diagrammi uniamo gli eventi e le operazioni di sistema, definiti nei capitoli precedenti attraverso gli SSD e i diagrammi di sequenza di progettazione, agli stati in cui l'applicazione si può trovare formalizzando quindi il ciclo di vita per gli elementi che compongono il sistema.

In particolare in questo capitolo approfondiamo il comportamento degli attuatori che il sistema è in grado di gestire.

9.2 Attuatori

Il framework PatrIot mette a disposizione delle classi che rappresentano diverse tipologie di attuatori le quali si differenziano per il loro comportamento, formalizzato attraverso una macchina a stati.
Una particolare caratteristica delle macchine a stati associate agli attuatori sta nelle transizioni, alle quali è associata una durata[1] del passaggio da uno stato all'altro.

Riportiamo in questa sezione i diagrammi delle macchine a stati che abbiamo associato agli attuatori del nostro sistema.

9.2.1 Attuatori per la gestione dell'illuminazione

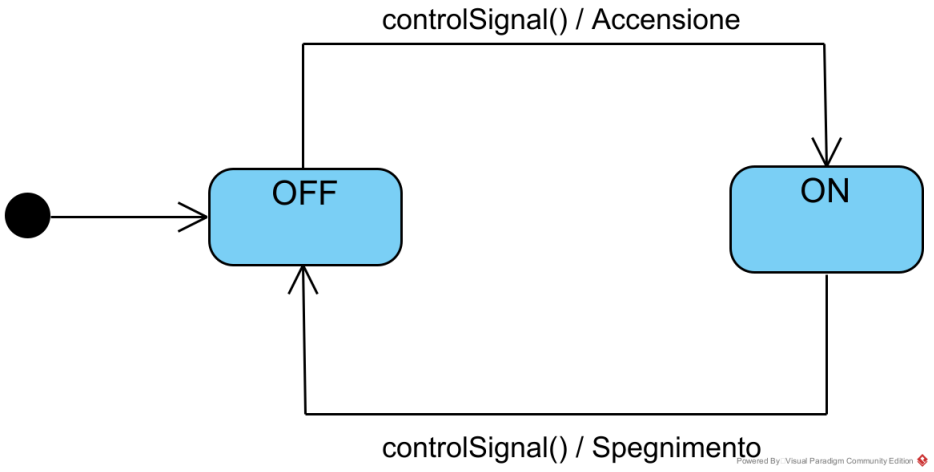

Immagine 9.1: Diagramma di Macchina a Stati per l'attuatore Luce

[1]Nei diagrammi riportati in questo capitolo abbiamo esplicitato per alcune transizioni la loro durata attraverso l'utilizzo dello strumento "Nota" offerto dal software Visual Paradigm.
Nei casi in cui la durata non viene esplicitata, le transizioni sono da considerarsi di durata 0ms; quindi istantanee.

9.2.2 Attuatori per la gestione della temperatura

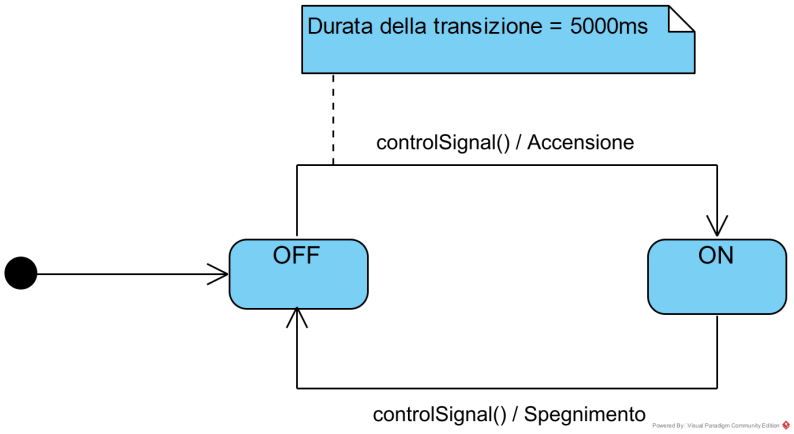

Immagine 9.2: Diagramma di Macchina a Stati per gli attuatori della gestione della Temperatura

9.2.3 Attuatori per la gestione delle finestre

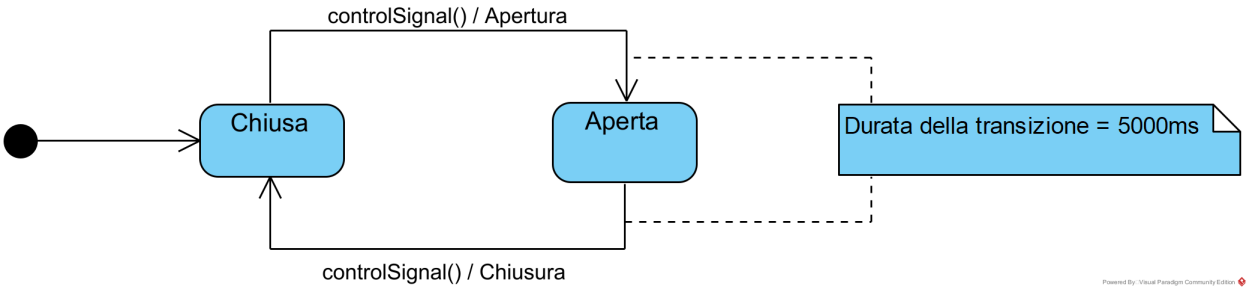

Immagine 9.3: Diagramma di Macchina a Stati per gli attuatori Finestra

9.2.4 Attuatori per la gestione dei dispositivi per la pulizia della casa

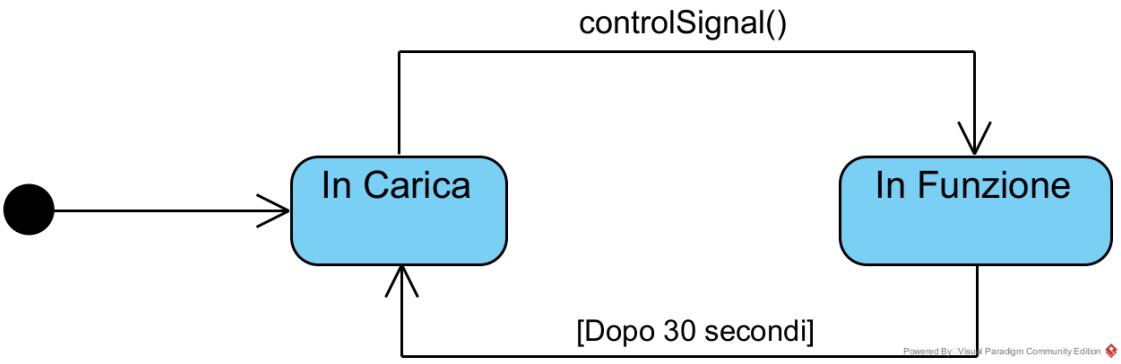

Immagine 9.4: Diagramma di Macchina a Stati per l'attuatore per la gestione della Pulizia della casa

Capitolo 10

Diagrammi di Attività

10.1 Introduzione

I diagrammi di attività offrono un'ulteriore visione dinamica del sistema rappresentando le attività svolte e le entità create durante i flussi di lavoro per la realizzazione delle funzionalità offerte dal sistema.

10.2 Gestione e analisi delle rilevazioni

Una sezione importante della nostra applicazione è la gestione delle rilevazioni effettuate dai sensori. Ciascuna di esse viene analizzata per verificare che all'interno della casa non vi sia una situazione che richiede l'intervento del sistema.
Riportiamo in questo capitolo i diagrammi che mostrano il workflow legato a queste operazioni di sistema.

10.2.1 Gestione Pericoli

Questo diagramma mostra la gestione delle rilevazioni effettuate da sensori di fumo e sensori per le fughe di gas. Gran parte del procedimento per la gestione di queste due aree di interesse è condiviso.

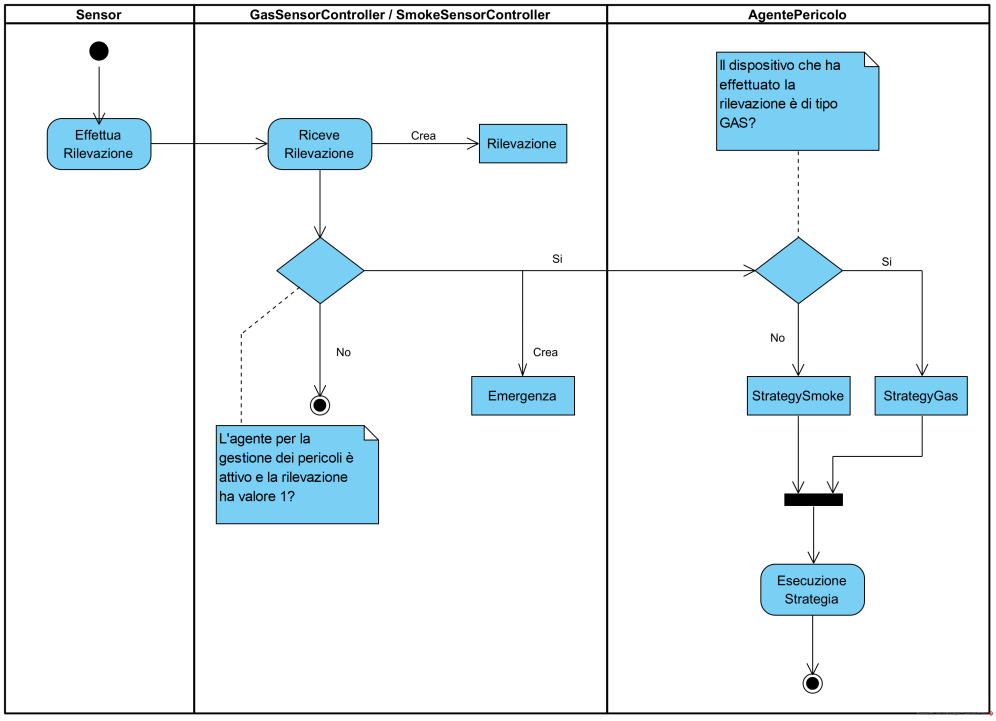

Immagine 10.1: Diagramma delle attività per la gestione dei pericoli

10.2.2 Gestione sensori di movimento

Questo diagramma mostra la gestione delle rilevazioni effettuate da sensori di movimento.
Queste rilevazioni possono essere gestite alternativamente dall'agente per l'allarme o da quello per le luci.

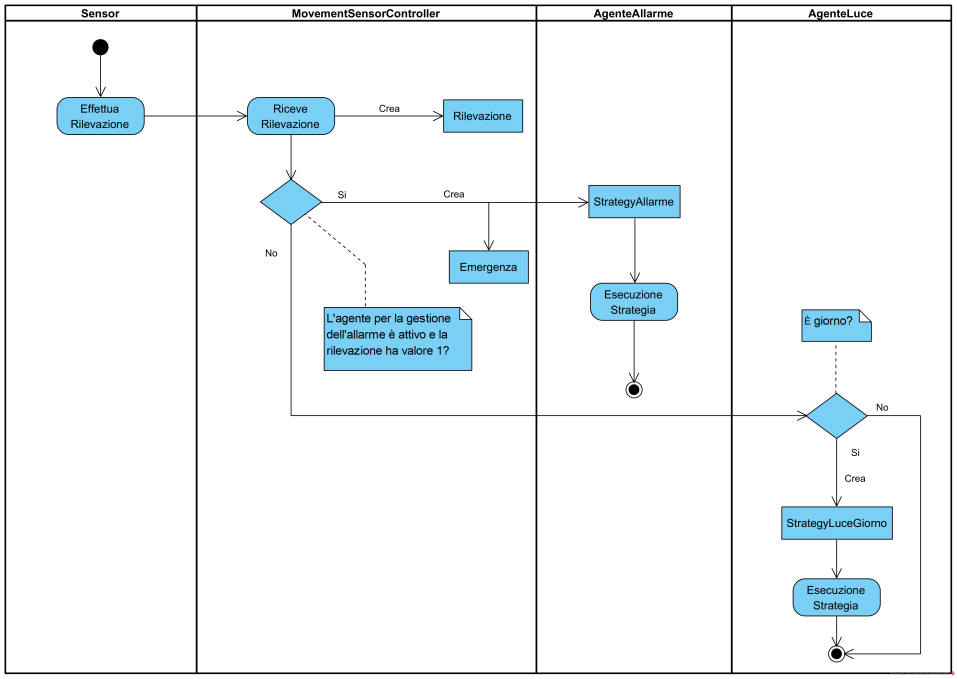

Immagine 10.2: Diagramma delle attività per la gestione dei sensori di movimento

10.2.3 Gestione sensori di temperatura

Questo diagramma mostra la gestione delle rilevazioni effettuate da sensori di temperatura.

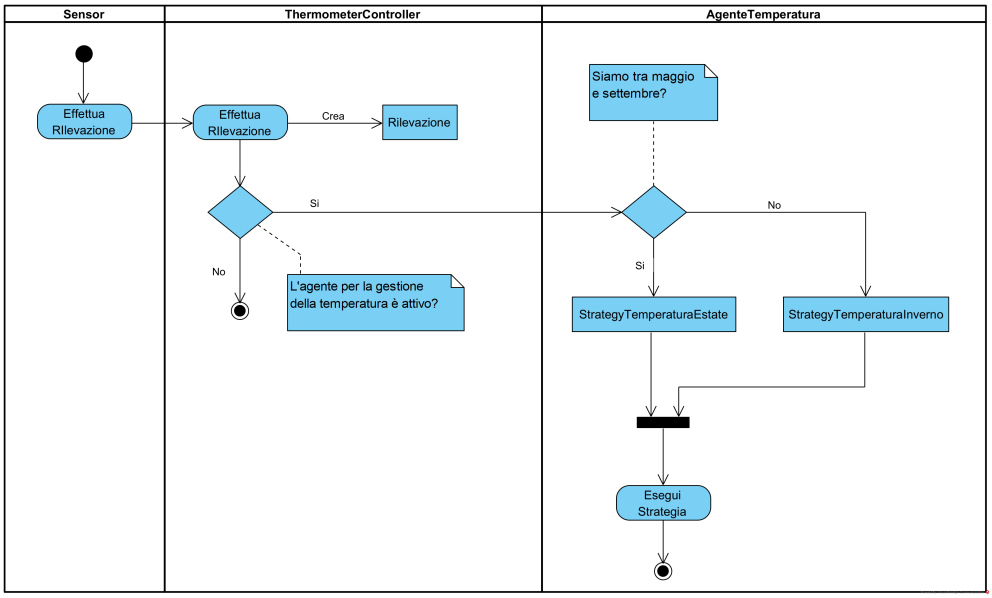

Immagine 10.3: Diagramma delle attività per la gestione dei sensori di temperatura

Capitolo 11

Analisi Finali sul Software

11.1 SonarQube

Terminata la fase di implementazione abbiamo effettuato diverse analisi del codice con SonarQube per verificare la presenza di Bug, Code Smell, Vulnerabilità o Problemi di Sicurezza.
I risultati delle analisi hanno evidenziato delle problematiche anche sul codice del framework PatrIot e, su indicazione della prof.ssa Pigazzini, abbiamo contrassegnato queste segnalazioni come *falsi positivi*.

11.2 Understand

Oltre alle verifiche descritte nella sezione precedente abbiamo analizzato le dipendenze tra i componenti software del nostro sistema attraverso il tool Understand.
Analizzando i diagrammi messi a disposizione dal tool abbiamo individuato alcune situazioni riconducibili a degli antipattern.

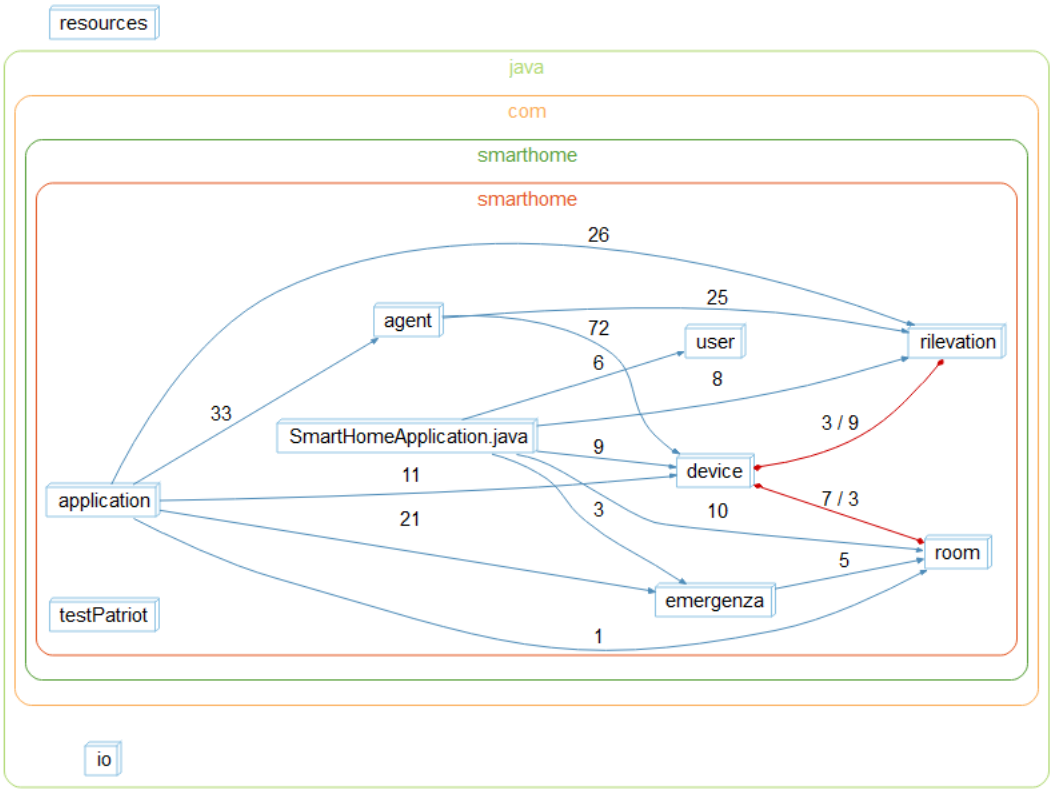

Immagine 11.1: Diagramma delle dipendenze tra i package del sistema

11.2.1 Actuator

Analizzando il diagramma Butterfly relativo alla classe Actuator possiamo notare che ci sono diversi classificatori, sia del framework patriot che del sistema smarthome, che dipendono dalla classe che astrae il concetto di attuatore.
Abbiamo quindi associato a questo scenario l'antipattern Global Butterfly.

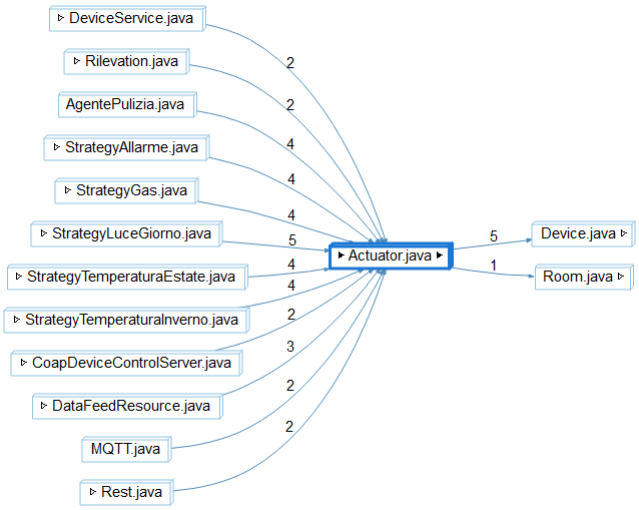

Immagine 11.2: Diagramma Butterfly per la classe Actuator

11.2.2 DeviceService

Analizzando il diagramma Butterfly relativo alla classe DeviceService, concentrandosi sugli oggetti software dipendenti, possiamo notare che ci sono diversi classificatori del sistema smarthome che fanno uso di un oggetto di tipo DeviceService per accedere al database.
Abbiamo quindi associato a questo scenario l'antipattern External Butterfly.

Immagine 11.3: Diagramma Butterfly per la classe DeviceService

11.2.3 Package agent

Analizzando il diagramma delle dipendenze tra i package che compongono il nostro sistema possiamo notare una forte correlazione tra il package agent e quello application.
Questo legame è mostrato anche nei diagrammi di attività, presenti in questa documentazione, dove viene descritto il flusso di lavoro con cui vengono realizzate le funzionalità del sistema.
Questa forte dipendenza è legata al fatto che i controller che si trovano nel package application, oltre a gestire le rilevazioni inviate dai sensori, creano gli oggetti di tipo agente per l'analisi delle rilevazioni. Da queste operazioni nasce il forte legame di dipendenza tra questi due package a cui abbiamo associato l'antipattern External Butterfly.

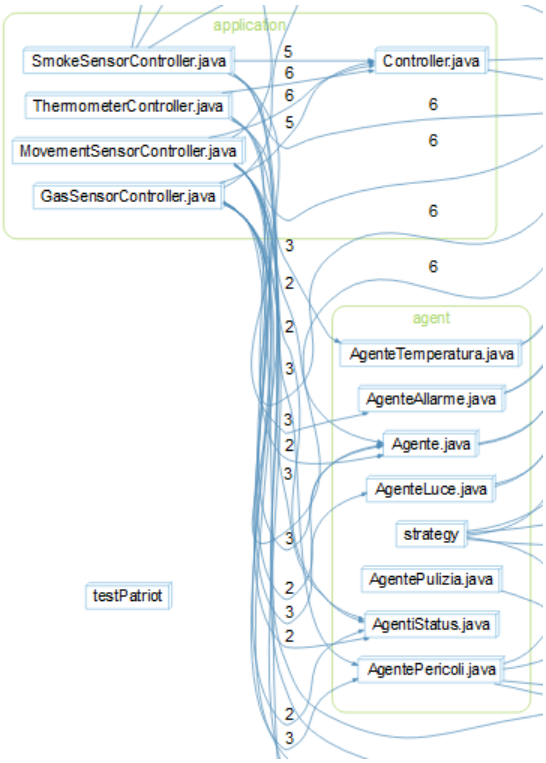

Immagine 11.4: Diagramma delle dipendenze tra i package del agent e application

Appendice A

Framework PatrIot

A.1 Scelta

Come indicato nei capitoli introduttivi di questo documento, gran parte della fase iniziale del progetto è stata dedicata allo studio del dominio dell'applicazione e del framework che avremmo utilizzato per il supporto del sistema nella simulazione e nella gestione del comportamento dei dispositivi.
Il framework iCasa, suggerito nel testo del progetto, si è dimostrato essere difficile da comprendere, vista la scarsa documentazione presente in rete, e poco supportato, dato che la versione più recente del software è stata rilasciata del 2015.
Abbiamo quindi ricercato una nuova piattaforma che offrisse funzionalità compatibili con il nostro progetto, individuando il framework PatIot. Parallelamente a questo primo periodo di studio del dominio e ricerca del framework avevamo avviato anche le fasi di ideazione, analisi dei requisiti e modellazione del sistema e della sua architettura.
L'utilizzo del framework PatrIot, meno completo di iCasa in termini di funzionalità offerte, ha inevitabilmente richiesto una revisione del lavoro fatto fino a quel momento e degli obiettivi che ci eravamo posti inizialmente, in funzione anche del poco tempo rimasto per la realizzazione del progetto[1].

A.2 Funzionalità

Abbiamo utilizzato i servizi offerti dal framework PatrIot per il supporto al sistema nella realizzazione delle seguenti funzionalità:

- **Comunicazione con i dispositivi:** Il framework offre la possibilità di definire per ciascun dispositivo che viene simulato un'interfaccia per la comunicazione bidirezionale con il sistema, attraverso il protocollo CoAP.
  Ad ogni device viene associata una label con cui è possibile identificarlo nelle chiamate alle API. Ai dispositivi di tipo sensore viene associato anche un endopoint al quale vengono inviate le rilevazioni generate facendo uso del protocollo HTTP.

- **Simulazione dei sensori:** Il framework permette di simulare il comportamento di un sensore nella generazione delle rilevazioni. Per ciascun sensore vengono definiti il DataFeed, ovvero l'insieme da cui viene preso il valore da inserire nelle rilevazioni, e la periodicità con cui il dispositivo invierà al sistema una nuova rilevazione.

- **Simulazione degli attuatori:** Il framework permette di simulare degli attuatori il cui comportamento è formalizzato attraverso delle macchine a stati.
  Per rappresentare al meglio tutte le tipologie di attuatori che il nostro sistema può gestire, abbiamo realizzato delle nuove classi all'interno delle quali abbiamo definito le macchine a stati che caratterizzano il comportamento dei diversi attuatori supportati dal nostro sistema.

[1] La distribuzione nel tempo delle attività descritte in questa sezione è facilmente visualizzabile attraverso il diagramma di Gantt che abbiamo realizzato.

Bibliografia

[1] Basman Alhafidh and William Allen. Design and simulation of a smart home managed by an intelligent self-adaptive system. *International Journal of Engineering Research and Applications*, 6:2248–962264, 08 2016.

Glossario

CoAP Constrained Application Protocol 23, 43

DAO Data Access Object 33

DB DataBase 22, 24, 27, 30, 32, 33

HTTP Hypertext Transfer Protocol 43

JSON JavaScript Object Notation 23, 26

SSD System Sequence Diagram (Diagramma di Sequenza di Sistema) 17, 18, 36