



Università degli Studi di Bergamo

SCUOLA DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

LVSEmergency

Progetto del corso Progettazione, Algoritmi e Computabilità

Prof.ssa
Patrizia Scandurra

Candidati
Davide Salvetti
Matricola 1057596

Lorenzo Leoni
Matricola xxxxxx

Matteo Verzeroli
Matricola 1057926



LVSEMERGENCY

Indice

1 Iterazione 0	4
1.1 Requisiti	4
1.2 Casi d'uso	5
1.2.1 Casi d'uso: Volontario	6
1.2.2 Casi d'uso: Caposquadra	6
1.2.3 Casi d'uso: Coordinatore	7
1.2.4 Diagramma dei casi d'uso	8
1.2.5 Priorità dei requisiti	9
1.3 Architettura del sistema	10
1.4 Toolchain e tecnologie utilizzate	13
2 Iterazione 1	14
2.1 Introduzione	14
2.2 System Diagram	15
2.3 Component Diagram	16
2.4 Data Class Diagram	17
2.5 Interface and Package Diagram	18
3 Iterazione 2	19
3.1 Introduzione	19
3.1.1 UC1 - Login	19
3.1.2 UC2 - Logout	20
3.1.3 UC3 - Visualizzazione informazioni account	20
3.1.4 UC4 - Visualizzazione informazioni squadra	21
3.1.5 UC6 - Segnalazione operatività	21
3.1.6 UC15 - Inserimento utente	21
3.1.7 UC16 - Cancellazione utente	22
3.1.8 UC17.1 - Creazione squadra	23
3.2 Component Diagram	24
3.3 Class Diagram	25
3.4 Interface and Package Diagram	26
3.5 Testing	27
3.5.1 Analisi statica	27
3.5.2 Analisi dinamica	27
3.5.3 Unit Test	29
3.6 Documentazione API	31

4 Iterazione 3	41
4.1 Introduzione	41
4.1.1 Data Collector	41
4.1.2 Data Analyzer	46
4.1.3 Test Data Analyzer	61
4.1.4 UC18.1 - Visualizzazione dati ambientali	64
4.1.5 UC18.2 - Visualizzazione allarmi	64
4.1.6 UC10 - Visualizzazione posizione real-time	64
4.2 Component Diagram	66
4.3 Class Diagram	67
4.4 Interface and Package Diagram	68
4.5 Testing	69
4.5.1 Analisi statica	69
4.5.2 Analisi dinamica	69
4.5.3 Unit Test	71
4.6 Documentazione API	75
5 Manuale Utente	81
6 Conclusioni	84
6.1 Sviluppi futuri	84
6.1.1 API Protezione Civile POP	85
6.1.2 API Terremoti	86
6.2 Approfondimento - Qt	87

Capitolo 1

Iterazione 0

1.1 Requisiti

Il cliente vuole realizzare una applicazione che permetta la gestione delle squadre della Protezione Civile. Gli attori in gioco sono i seguenti:

- **volontario**: colui che opera durante gli interventi;
- **caposquadra**: responsabile di un gruppo di volontari;
- **coordinatore**: amministratore del sistema, si occupa dell'inserimento di nuovi volontari e della gestione delle squadre.

I volontari sono organizzati in squadre ed ogni squadra ha un caposquadra che coordina le attività. Ogni squadra si occupa degli interventi all'interno di una specifica area.

Per ogni attore devono essere memorizzate le informazioni anagrafiche di base (nome, cognome, CF, residenza, ...). Per volontari e capisquadra dovrà essere memorizzato anche lo stato operativo: un volontario (o un caposquadra) può dichiararsi operativo solo quando sta partecipando ad un intervento; quando un volontario (o un caposquadra) è operativo, tutti i membri della squadra potranno visualizzarne la posizione sulla mappa.

Gli interventi possono essere di due tipi: programmati oppure emergenze. È compito del caposquadra creare un nuovo intervento. Per ogni intervento bisogna memorizzare alcune informazioni (tipo di intervento, descrizione dell'intervento, materiali utilizzati durante l'intervento, partecipanti, ora, luogo, ...) e poi, al termine, generare un report (in formato PDF) con un riassunto dei dati ed eventuali foto scattate.

I volontari potranno inserire delle ore di reperibilità. Nel momento in cui viene creato un intervento programmato o un intervento di emergenza, l'applicazione, sulla base delle reperibilità e della difficoltà dell'intervento, creerà una squadra temporanea formata dai volontari disponibili, ai quali verrà inviata una notifica con le informazioni dell'intervento. A questo punto i volontari parteciperanno all'intervento e potranno documentare tutto ciò che è stato fatto, caricare foto ed inserire i materiali utilizzati.

Il coordinatore ha il compito di inserire nuovi volontari nel sistema ed assegnarli ad una squadra esistente, oppure può creare nuove squadre. Nel momento in cui il coordinatore crea una nuova squadra, dovrà inserire anche la zona a cui la squadra verrà assegnata.

L'applicazione deve fornire agli operatori la possibilità di visualizzare le informazioni di bollettini meteo, terremoti, allerte della protezione civile nazionali ed eventi particolari provenienti da stazioni installate sul territorio. Tali informazioni vengono recuperate tramite le seguenti API:

- APRS.FI: sito utilizzato dai radioamatori per la condivisione di dati open. Il sito fornisce alcune API REST da cui è possibile richiedere informazioni su stazioni meteorologiche pubbliche;

- OpenWeatherMap: il sito permette di scaricare informazioni meteorologiche, previsioni e dati sulla qualità dell'aria in tutta Italia tramite delle API REST;
- INGV Earthquake event: permette di ricevere informazioni sui terremoti che si sono verificati in una particolare area geografica.
- Protezione Civile POP: web app che fornisce delle API tramite cui è possibile scaricare i bollettini giornalieri della Protezione Civile (recuperati dalla repository Github della Protezione Civile italiana) riguardo al rischio idraulico, idrogeologico e temporali in una certa zona;
- Recupero di dati da stazioni personalizzate posizionate da una squadra: la stazione fornirà una API dalla quale poter recuperare dati di interesse (per esempio dati meteorologici o livello di torrenti).

L'applicazione analizzerà i dati provenienti dalle stazioni meteorologiche per effettuare previsioni a breve termine e avvisare le squadre in caso di particolari fenomeni tramite degli allarmi (presenza di nebbia o brina, maltempo in avvicinamento/allontanamento, livelli di pioggia elevati, livello dell'acqua di un fiume elevato).

Le stazioni personalizzate installate dalla squadra potranno essere realizzate tramite prototipi Arduino/Raspberry e dei sensori, ed esporranno una API REST alla quale richiedere le informazioni, secondo un protocollo definito. Ad esempio, si potrà realizzare un sistema di monitoraggio del livello dell'acqua di fiumi, per essere avvisati in caso di straripamento, oppure dati meteorologici (per esempio temperatura, umidità, vento) per avere informazioni dettagliate su un'area di particolare interesse (per esempio aree protette o risorse naturali, dove non sono già presenti delle stazioni).

Il caposquadra potrà decidere a quali servizi di notifica, riguardanti la zona della sua squadra, vuole sottoscriversi. Sarà possibile, infatti, richiedere di essere informati sulle condizioni meteo ed altre allerte generate da un'analisi dei dati, effettuata dall'applicazione, di una o più stazioni accessibili su APRS.FI, specificando il codice della centralina desiderata. Oppure, potrà scegliere di ricevere i bollettini meteo basati sulle previsioni di OpenWeatherMap (che fornisce previsioni più precise ma riguardanti una zona più generica rispetto all'area operativa della squadra). Oppure, in base alla località assegnata alla squadra, potrà richiedere di essere avvisato in caso di terremoti in quella zona. Infine, si potrà richiedere di essere avvisati sui bollettini giornalieri emessi dalla Protezione Civile nazionale. Infine, analizzando i dati provenienti dalle stazioni posizionate delle squadre potranno essere generati allarmi che saranno notificati al caposquadra responsabile della zona.

All'interno dell'applicazione sarà possibile visualizzare il luogo degli interventi o la posizione dei volontari operativi su una mappa grazie ai servizi di *OpenStreetMap*. Inoltre, in fase di registrazione di un area potranno essere utilizzate delle API (fornite dal sito <https://comuni-ita.herokuapp.com/>) che permettono di ricevere alcune informazioni precise (posizione, cap, codice istat, etc.) sui comuni italiani.

L'applicazione deve poter essere eseguita sia su PC (Windows e MacOS) sia su dispositivi mobili (Android e iOS).

1.2 Casi d'uso

Di seguito vengono riportati brevemente i casi d'uso raggruppati in base all'attore coinvolto. Gli attori in gioco sono i seguenti:

- volontario;
- caposquadra;
- coordinatore.

1.2.1 Casi d'uso: Volontario

- **UC1 - Login:** come volontario, voglio poter accedere al mio profilo personale.
- **UC2 - Logout:** come volontario, voglio poter disconnettere il mio profilo personale dall'applicazione.
- **UC3 - Visualizzazione informazioni account:** come volontario, voglio poter accedere alle informazioni relative al mio account.
- **UC4 - Visualizzazione informazioni squadra:** come volontario, voglio poter accedere alle informazioni relative alla mia squadra.
- **UC5 - Gestione reperibilità:** come volontario, voglio poter indicare e modificare le mie ore di reperibilità.
- **UC6 - Segnalazione operatività:** come volontario, voglio poter segnalare quando sono operativo e mi trovo sul luogo dell'intervento.
- **UC7 - Visualizzazione intervento di emergenza:** come volontario, voglio poter visualizzare le informazioni relative ad un intervento di emergenza a cui sono stato assegnato.
- **UC8 - Visualizzazione intervento programmato:** come volontario, voglio poter visualizzare le informazioni relative ad un intervento programmato a cui sono stato assegnato.
- **UC9 - Inserimento informazioni intervento:** come volontario, voglio poter arricchire la documentazione relativa all'intervento, programmato oppure di emergenza, a cui sono stato assegnato, aggiungendo foto e osservazioni.
- **UC10 - Visualizzazione posizione real-time:** come volontario, voglio poter visualizzare su una mappa la posizione in tempo reale dei colleghi operativi appartenenti alla mia squadra durante un intervento.
- **UC18 - Visualizzazione informazioni zona:** come volontario, voglio poter consultare le informazioni riguardanti il meteo, i terremoti, la qualità dell'aria, le stazioni personalizzate e gli allarmi relativi alla zona assegnata alla mia squadra.

1.2.2 Casi d'uso: Caposquadra

Il caposquadra include tutti i casi d'uso del volontario. In aggiunta si hanno i seguenti:

- **UC11 - Gestione intervento di emergenza:** come caposquadra, voglio poter creare, modificare o eliminare un intervento di emergenza.
- **UC12 - Gestione intervento programmato:** come caposquadra, voglio poter creare, modificare o eliminare un intervento programmato.
- **UC13 - Gestione report intervento di emergenza:** come caposquadra, voglio poter visualizzare e modificare i report riguardanti gli interventi di emergenza creati da me.
- **UC14 - Gestione report intervento programmato:** come caposquadra, voglio poter visualizzare e modificare i report riguardanti gli interventi programmati creati da me.
- **UC19 - Notifiche allarmi zona:** come caposquadra, voglio poter decidere su quali allarmi relativi alla mia zona essere notificato.
- **UC20 - Gestione informazioni relative alla zona:** come caposquadra, voglio poter inserire, modificare ed eliminare i dati identificativi delle stazioni APRS da cui ricevere i dati meteorologici.

1.2.3 Casi d'uso: Coordinatore

- **UC1 - Login:** come coordinatore, voglio poter accedere al mio profilo personale.
- **UC2 - Logout:** come coordinatore, voglio poter disconnettere il mio profilo personale dall'applicazione.
- **UC3 - Visualizzazione informazioni account:** come coordinatore, voglio poter accedere alle informazioni relative al mio account.
- **UC15 - Inserimento utente:** come coordinatore, voglio poter inserire un nuovo utente.
- **UC16 - Cancellazione utente:** come coordinatore, voglio poter cancellare un utente.
- **UC17 - Gestione squadre:** come coordinatore, voglio poter creare, modificare, eliminare le squadre e assegnare a ognuna di esse un caposquadra.

1.2.4 Diagramma dei casi d'uso

In figura 1.1 è stato riportato il diagramma dei casi d'uso. In questo diagramma sono stati inseriti i casi d'uso descritti in precedenza mettendoli in correlazione con il corrispettivo attore. È importante notare che la figura del *caposquadra* è una specializzazione del *volontario* e di conseguenza eredita tutti i suoi casi d'uso, oltre ad implementarne altri personalizzati. I componenti in grigio sono invece gli attori secondari, i quali sono esterni al sistema e servono per implementare alcune funzionalità particolari.

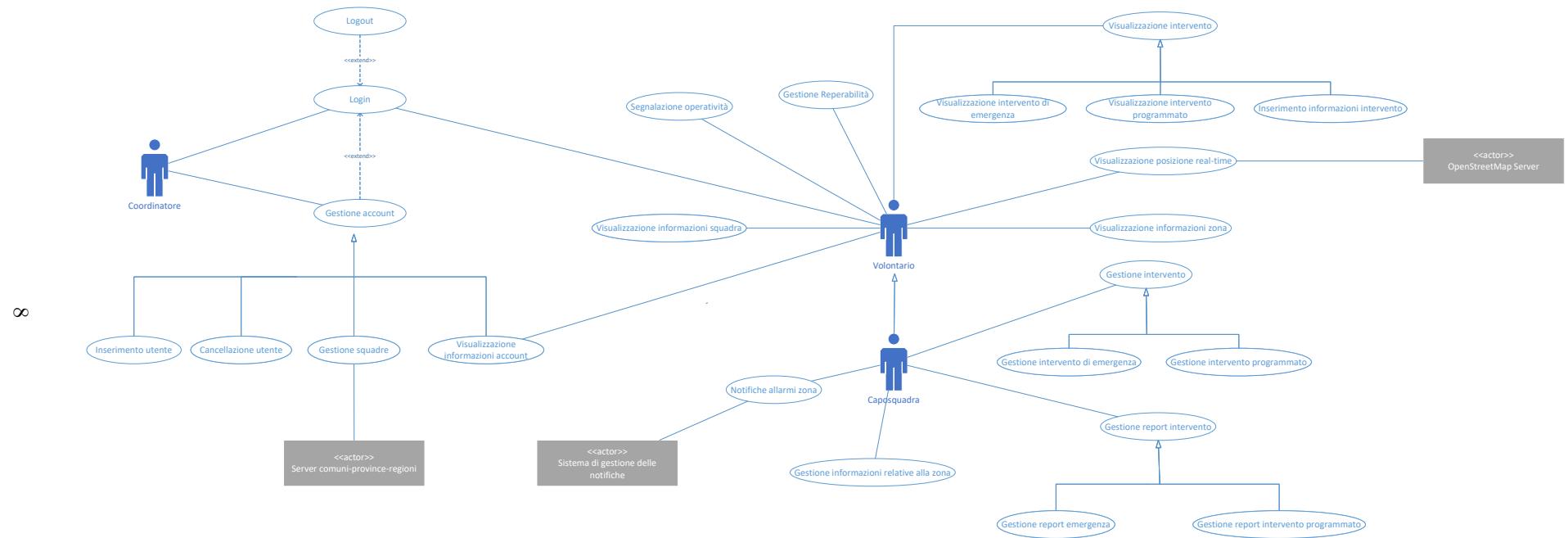


Figura 1.1: Diagramma dei casi d'uso.

1.2.5 Priorità dei requisiti

Coda ad alta priorità

I casi d'uso con priorità più alta sono quelli necessari alla popolazione del database, alla gestione dell'account utente ed alla visualizzazione delle informazioni sulla zona.

Codice	Caso d'uso
UC1	Login
UC2	Logout
UC3	Visualizzazione informazioni account
UC4	Visualizzazione informazioni squadra
UC15	Inserimento utente
UC16	Cancellazione utente
UC17	Gestione squadre
UC18	Visualizzazione informazioni zona

Tabella 1.1: Coda ad alta priorità.

Coda a media priorità

I casi d'uso con priorità media sono quelli relativi alla gestione degli interventi.

Codice	Caso d'uso
UC6	Segnalazione operatività
UC7	Visualizzazione intervento di emergenza
UC8	Visualizzazione intervento programmato
UC9	Inserimento informazioni intervento
UC11	Gestione intervento di emergenza
UC12	Gestione intervento programmato
UC13	Gestione report intervento di emergenza
UC14	Gestione report intervento programmato
UC20	Gestione informazioni relative alla zona

Tabella 1.2: Coda a media priorità.

Coda a bassa priorità

I casi d'uso con bassa priorità sono quelli relativi a funzionalità aggiuntive non fondamentali.

Codice	Caso d'uso
UC5	Gestione reperibilità
UC10	Visualizzazione posizione real-time
UC19	Notifiche allarmi zona

Tabella 1.3: Coda a bassa priorità.

1.3 Architettura del sistema

L’architettura del sistema è stata formalizzata attraverso due *deployment diagram* differenti: il primo, riportato in figura 1.2, rappresenta il sistema tramite una notazione a stile libero, mentre il secondo, riportato in figura 1.3, rappresenta il sistema tramite lo stile UML. In particolare, in entrambi i diagrammi è possibile identificare un pattern architettonicale *three-tier*. Il *presentation layer* è costituito dalle applicazioni lato client che vengono utilizzate dagli utenti e che hanno solamente il compito di recuperare le informazioni dal sistema e visualizzarle. Nell’*application layer*, invece, ci sono due sistemi distinti:

- l’applicativo **LVSEmergency**, che si occupa della gestione di richieste HTTP/REST, le quali permettono l’interazione con il client. Il server sarà progettato secondo lo stile esagonale a microservizi;
- due componenti autonome: un **Data Collector** che si occupa di recuperare informazioni dai *data server* tramite API REST e di inserirle all’interno del database, che costituisce il *data layer*, ed un **Data Analyzer** che si occupa di effettuare delle analisi sui dati acquisiti con lo scopo di identificare possibili situazioni di allarme. È bene notare che queste due applicazioni non costituiscono una vera e propria *Big Data Pipeline* ma ne rappresentano una versione semplificata, in quanto vengono effettuate solo la collezione e l’analisi dei dati raccolti.

Nel *data layer* troviamo invece un database all’interno del quale sono inseriti tutti i dati necessari per il funzionamento dell’applicazione.

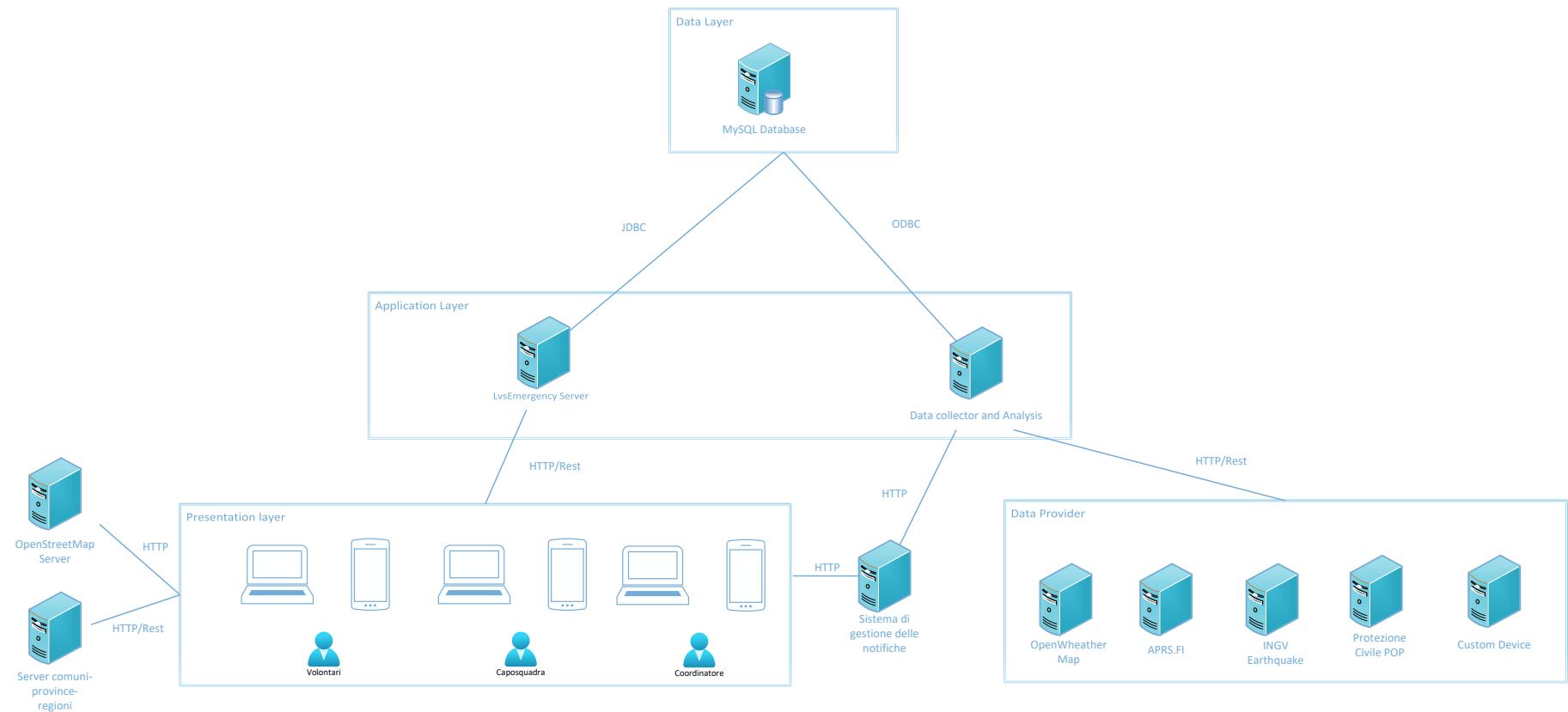


Figura 1.2: Deployment diagram in stile libero.

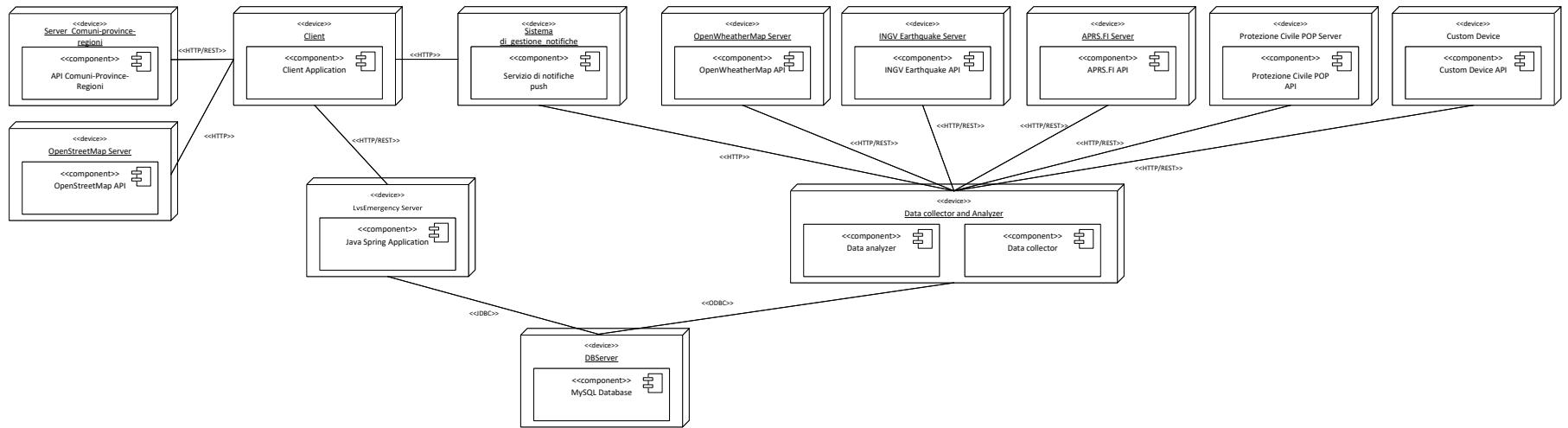


Figura 1.3: Deployment diagram in stile UML.

1.4 Toolchain e tecnologie utilizzate

Per la realizzazione del sistema verranno utilizzati i seguenti strumenti:

- **Modellazione:**
 - diagramma dei casi d'uso, deployment diagram, component diagram, class diagram, diagrammi di flusso, diagramma entità-relazione: Visio.
- **Implementazione Applicazione Client:**
 - Linguaggio di programmazione: C++.
 - IDE: Qt Creator 6.0.2.
 - Interfaccia grafica: QML.
 - Analisi statica: CppCheck.
 - Analisi dinamica: Qt Test.
- **Implementazione Web Server:**
 - Linguaggio di programmazione: Java.
 - IDE: Eclipse.
 - Framework: Spring.
 - Analisi statica: STAN4J.
 - Analisi dinamica: JUnit.
 - Deployment: Azure Spring Cloud.
- **Implementazione Data Analyzer e Data Fetcher:**
 - Linguaggio di programmazione: Python.
 - IDE: Visual Studio Code.
 - Server: Raspberry Pi 4
 - Analisi dinamica: unittest
- **Implementazione Database:**
 - Tipologia: relazionale.
 - Database: MySQL.
 - Provider: Azure.
- **Documentazione, versioning e organizzazione del team:**
 - Documentazione: Latex, con scrittura tramite editor TexStudio.
 - Versioning: GitHub.
 - Git client: Sourcetree.
 - Organizzazione del Team: Microsoft Teams.

Capitolo 2

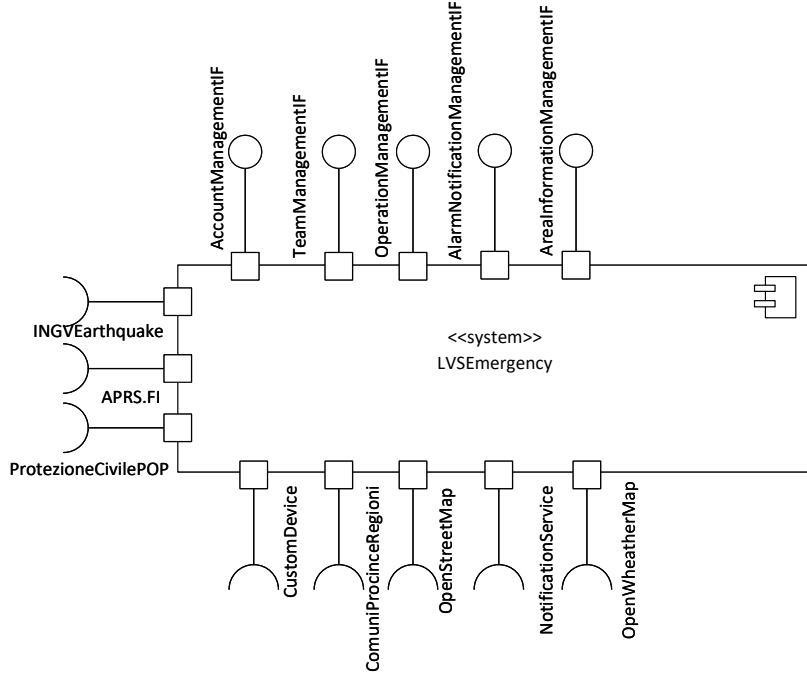
Iterazione 1

2.1 Introduzione

Nell'iterazione 1 è stata perfezionata la specifica dei componenti progettati durante l'iterazione 0 in modo da definire meglio la struttura dell'applicazione. Si è anche iniziato ad impostare il progetto Spring in Eclipse e quello lato client in Qt, costruendo lo scheletro delle applicazioni tramite la specifica dei differenti package che verranno popolati nelle seguenti iterazioni. Inoltre, è stata creata un'istanza di un database *MySQL* su Azure, così da poter iniziare a definire il database, componente fondamentale per l'implementazione del progetto.

2.2 System Diagram

Nella figura 2.1 sono rappresentate tutte le interfacce richieste ed esposte dal nostro sistema. Inoltre, i casi d'uso sono stati raggruppati, indicando il nome dell'interfaccia che permetterà di implementare il particolare caso d'uso.



- G1: [UC1, UC2, UC3, UC4, UC5, UC6, UC10] AccountManagementIF
- G2: [UC15, UC16, UC17] TeamManagementIF
- G3: [UC7, UC8, UC9, UC11, UC12, UC13, UC14] OperationManagementIF
- G4: [UC19] AlarmNotificationManagementIF
- G5: [UC18, UC20] ArealInformationManagementIF

Figura 2.1: System Diagram.

2.3 Component Diagram

L'applicazione presente sul server *LVSEmergency* è una *Spring Boot Application* tramite cui vengono gestite le richieste HTTP/REST inviate dalla *Client Application*. Ogni interfaccia esposta dal server *LVSEmergency* viene gestita da un componente autonomo che interagisce con il database.

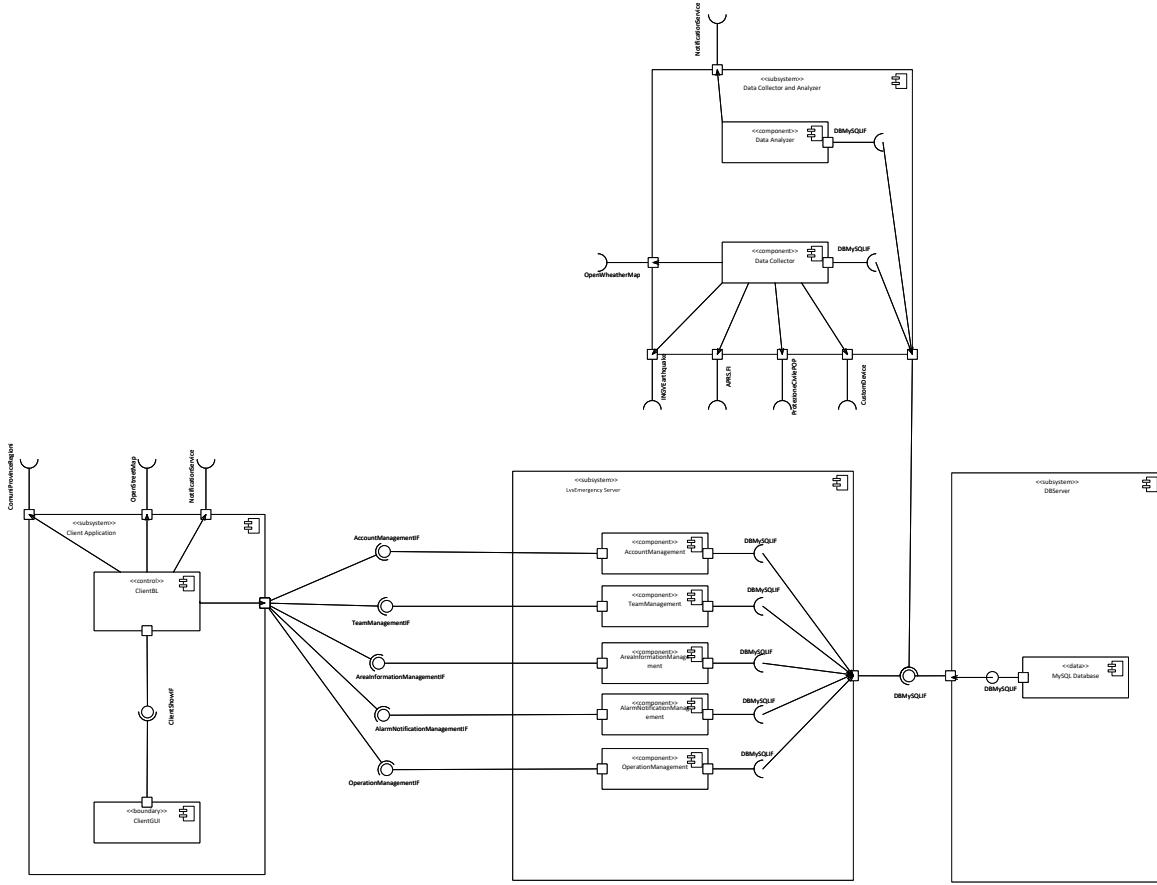


Figura 2.2: Component Diagram.

2.4 Data Class Diagram

Di seguito viene riportato il diagramma delle classi che rappresenta la struttura del database e delle entità presenti nell'applicazione.



Figura 2.3: Data Class Diagram.

2.5 Interface and Package Diagram

Nella figura 2.4 è rappresentato il diagramma delle interfacce e dei package che riassume la struttura logica dell'applicazione e delle interfacce che verranno implementate nelle successive iterazioni.

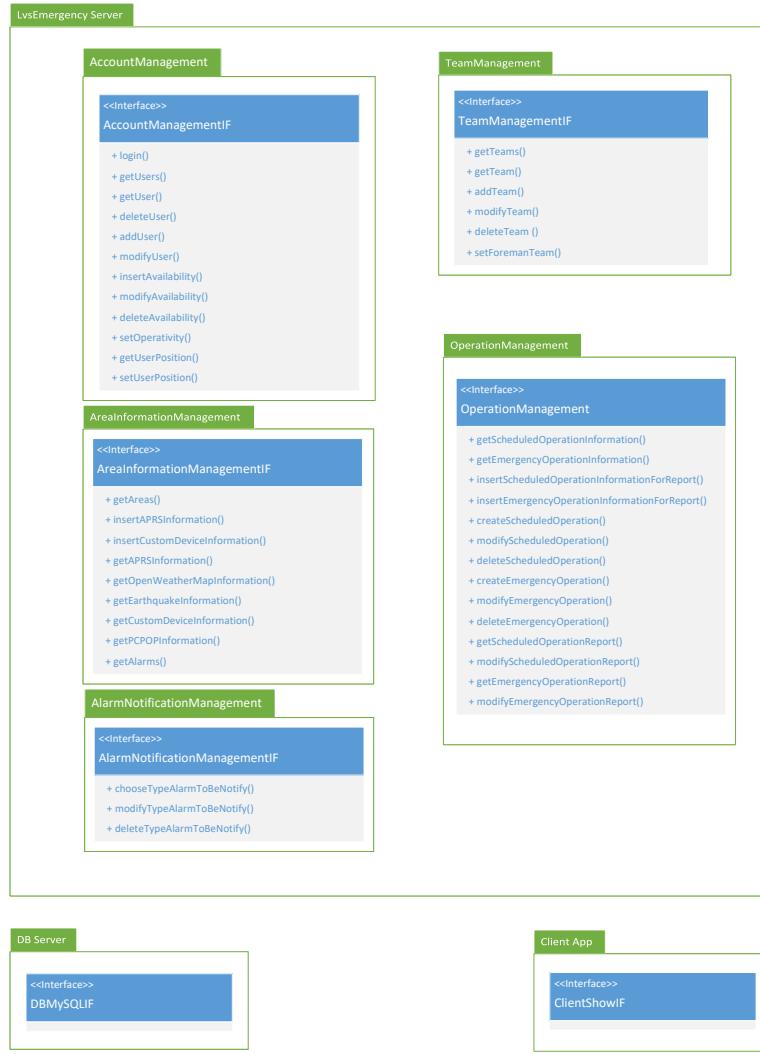


Figura 2.4: Interface and Package Diagram.

Capitolo 3

Iterazione 2

3.1 Introduzione

Nella seconda iterazione sono stati implementati i seguenti casi d'uso:

- UC1 - Login;
- UC2 - Logout;
- UC3 - Visualizzazione informazioni account;
- UC4 - Visualizzazione informazioni squadra;
- UC6 - Segnalazione operatività;
- UC15 - Inserimento utente;
- UC16 - Cancellazione utente;
- UC17 - Gestione squadre [astratto]:
 - UC17.1 - Creazione squadra;

3.1.1 UC1 - Login

Per l'autenticazione è stato utilizzato il metodo *Basic Access Authentication*: si tratta di una tecnica che non necessita dell'utilizzo di cookie o di mantenere una sessione tra client e server, ma utilizza gli *header* HTTP per fornire le informazioni di accesso. In particolare, *username* e *password* vengono codificate con *base64* e vengono trasmesse nell'*header* ogni volta che viene chiamata un API. Il sistema poi verifica che lo *username* e la *password* presenti nell'*header* HTTP appartengano effettivamente ad un utente presente nel database prima di elaborare la richiesta.

Breve descrizione: l'utente compila il form per eseguire il login: in caso di credenziali corrette il sistema consente l'accesso ai servizi, altrimenti notifica l'utente della non correttezza delle credenziali.

Attori coinvolti: volontario, caposquadra, coordinatore, sistema.

Precondizione: l'utente è registrato nel sistema.

Postcondizione: l'utente accede alla *dashboard* (in caso le credenziali siano corrette) oppure viene notificato dell'incorrectezza delle credenziali.

Procedimento:

1. il sistema richiede all'utente le informazioni di accesso (username e password).
2. l'utente inserisce le informazioni di accesso.
3. il sistema controlla le informazioni fornite.
4. le informazioni sono corrette. [E1: le informazioni sono sbagliate].
5. l'utente viene loggato nel sistema.

Eccezioni:

- E1:

1. le informazioni sono sbagliate.
2. il sistema comunica all'utente che le informazioni inserite non sono corrette.
3. ritorno al passo 1 del "Procedimento".

3.1.2 UC2 - Logout

Grazie alla *Base Access Authentication*, il logout dell'utente consiste nel rimandarlo alla pagina di login per richiedere nuovamente *username* e *password*. Infatti, non c'è nessuna sessione tra client e server, quindi questo caso d'uso viene gestito lato client chiedendo all'utente di autenticarsi di nuovo.

Breve descrizione: il sistema esegue il logout dell'utente.

Attori coinvolti: volontario, caposquadra, coordinatore, sistema.

Precondizione: l'utente è loggato nel sistema.

Postcondizione: l'utente viene rimandato alla *login page* e non è più loggato nel sistema.

Procedimento:

1. l'utente richiede al sistema di eseguire il logout.
2. il sistema effettua il logout dell'utente.

3.1.3 UC3 - Visualizzazione informazioni account

Breve descrizione: l'utente visualizza le informazioni del suo account.

Attori coinvolti: volontario, caposquadra, coordinatore, sistema.

Precondizione: l'utente è loggato nel sistema. L'utente è nella pagina "Informazioni".

Postcondizione: il sistema mostra le informazioni dell'account dell'utente.

Procedimento:

1. il sistema mostra le informazioni dell'account dell'utente.

3.1.4 UC4 - Visualizzazione informazioni squadra

Breve descrizione: l'utente visualizza le informazioni della squadra a cui è stato assegnato.

Attori coinvolti: volontario, caposquadra, sistema.

Precondizione: l'utente è loggato nel sistema. L'utente è nella pagina "Informazioni".

Postcondizione: il sistema mostra le informazioni della squadra a cui l'utente è assegnato.

Procedimento:

1. il sistema mostra le informazioni della squadra a cui l'utente è assegnato tra cui:

- Nome squadra;
- Nome area;
- Numero dei membri della squadra;
- Informazioni sugli utenti appartenenti alla squadra. Per ogni utente viene mostrato:
 - Nome;
 - Cognome;
 - Ruolo;
 - Cellulare;
 - Email.

3.1.5 UC6 - Segnalazione operatività

Breve descrizione: l'utente segnala lo stato di operatività ("Attivo" o "Inattivo").

Attori coinvolti: volontario, caposquadra, sistema.

Precondizione: l'utente è loggato nel sistema.

Postcondizione: il sistema salva lo stato di operatività dell'utente.

Procedimento:

1. l'utente clicca sul pulsante per cambiare il suo stato di operatività.
2. il sistema salva lo stato di operatività e lo mostra all'utente.

3.1.6 UC15 - Inserimento utente

Breve descrizione: il coordinatore compila il form per inserire un nuovo volontario di cui ha ricevuto una richiesta di iscrizione.

Attori coinvolti: coordinatore, sistema.

Precondizione: il coordinatore è loggato nel sistema. Il coordinatore è nella pagina "Inserisci utente".

Postcondizione: il volontario viene inserito nel database e può accedere ai servizi.

Procedimento:

1. il coordinatore fornisce le seguenti informazioni nel form per la registrazione di un nuovo volontario:
 - Username;
 - Nome;
 - Cognome;
 - Password;
 - Sesso;
 - Codice fiscale;
 - Indirizzo;
 - Cellulare;
 - Email;
 - Ruolo: nel caso il ruolo selezionato sia "Caposquadra" verranno mostrate solo le squadre a cui non è ancora stato assegnato un caposquadra;
 - Squadra.
2. il coordinatore clicca il pulsante "Inserisci utente".
3. il sistema verifica che tutti i campi siano stati compilati [E1; ci sono dei campi vuoti].
4. il sistema verifica che username e codice fiscale non siano già associati ad altri utenti.
5. il sistema aggiunge l'utente con i rispettivi dati nel database [E2: i dati sono già stati associati ad un altro utente presente nel database].
6. se l'utente inserito ha il ruolo di "Caposquadra", il sistema imposta l'utente come caposquadra della squadra selezionata.

Eccezioni:

- E1:
 1. i dati sono già associati ad un altro utente presente nel database.
 2. il sistema comunica al coordinatore che i dati sono già associati ad un altro utente.
 3. ritorno al passo 1 del "Procedimento".
- E2:
 1. i dati sono già associati ad un altro utente presente nel database.
 2. il sistema comunica al coordinatore che i dati sono già associati ad un altro utente.
 3. ritorno al passo 1 del "Procedimento".

3.1.7 UC16 - Cancellazione utente

Breve descrizione: il coordinatore cancella un utente dal sistema.

Attori coinvolti: coordinatore, sistema.

Precondizione: il coordinatore è loggato nel sistema. L'utente che deve essere cancellato è inserito nel sistema. Il coordinatore è nella pagina "Cancella utente".

Postcondizione: l'utente viene cancellato dal sistema.

Procedimento:

1. il sistema mostra tutti gli utenti registrati tranne i coordinatori che non possono essere cancellati per motivi di sicurezza.
2. il coordinatore preme il pulsante per cancellare un utente.
3. l'utente viene cancellato.

3.1.8 UC17.1 - Creazione squadra

Breve descrizione: il coordinatore crea una nuova squadra.

Attori coinvolti: coordinatore, sistema.

Precondizione: il coordinatore è loggato nel sistema.

Postcondizione: la nuova squadra è inserita nel database.

Procedimento:

1. il coordinatore inserisce le informazioni richieste per la creazione di una squadra:
 - il nome della squadra.
 - l'area di competenza della squadra.
2. il coordinatore preme il pulsante "Crea squadra".
3. il sistema verifica che siano stati inseriti tutti i campi [E1: ci sono dei campi vuoti].
4. il sistema verifica che il nome della squadra non sia già stato assegnato ad altre squadre [E2: il nome della squadra è già stato assegnato ad un'altra squadra].
5. il sistema inserisce la nuova squadra nel database.

Eccezioni:

- E1:
 1. ci sono dei campi vuoti.
 2. il sistema comunica al coordinatore che ci sono dei campi vuoti.
 3. ritorno al passo 1 del "Procedimento".
- E2:
 1. il nome della squadra è già stato assegnato.
 2. il sistema comunica al coordinatore che il nome della squadra è già in uso.
 3. ritorno al passo 1 del "Procedimento".

3.2 Component Diagram

Nella figura 3.1 è riportato il diagramma a componenti relativo ai casi d'uso sviluppati nell'iterazione 2. Per l'implementazione dei microservizi REST è stato utilizzato un modello esagonale: ogni componente all'interno del server *LVSEmergency* è costituito da un *REST Controller*, un'*ApplicationService* e una *Repository*. Lato client il progetto è stato strutturato in modo molto simile al server: è stato sviluppato un controller per ogni interfaccia esposta, con il compito di invocare e gestire le API fornite da quella specifica interfaccia.

da rileggere

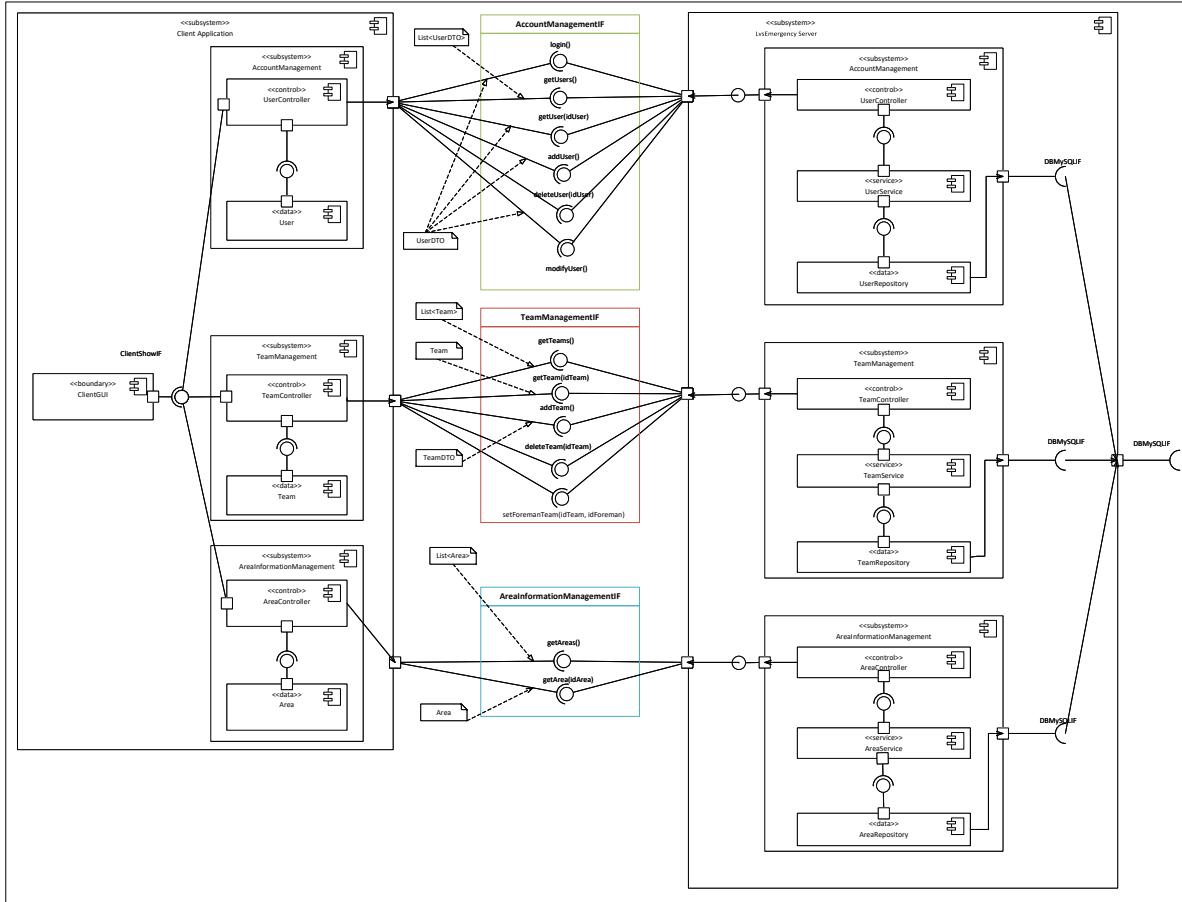


Figura 3.1: Component Diagram.

3.3 Class Diagram

Di seguito sono riportate le specifiche delle strutture dati `UserDTO` e `TeamDTO` (Fig.3.2) inserite nel server. Esse sono utilizzate per definire il formato dei dati scambiati tra l'applicazione client e applicazione server. In particolare, grazie alla direttiva `@JsonProperty(access = JsonProperty.Access.WRITE_ONLY)`, il campo `password` presente in `UserDTO` non viene mai inviato dal server al client, ma viene inserito dall'app client durante l'inserimento di un nuovo utente.



Figura 3.2: Class Diagram `UserDTO` e `TeamDTO`

3.4 Interface and Package Diagram

Nella Fig.3.3 è rappresentato il diagramma delle interfacce e dei package, nel quale è stata specificata anche la firma delle interfacce introdotte nell'iterazione 2.

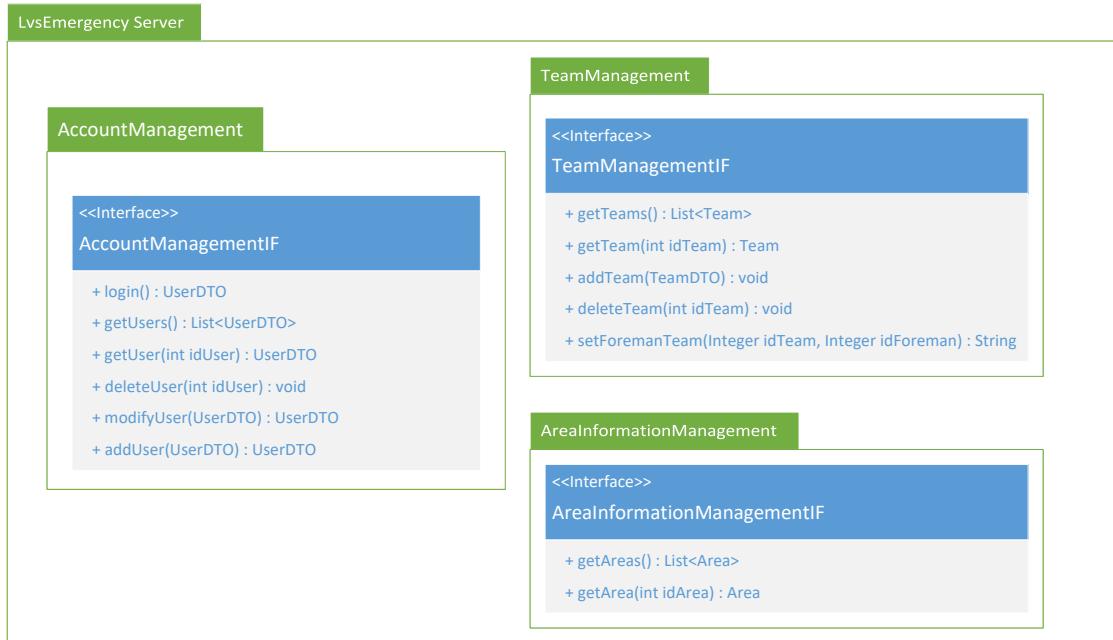


Figura 3.3: Interface and Package Diagram.

3.5 Testing

3.5.1 Analisi statica

Per l'analisi statica del codice Java è stato utilizzato il tool STAN4J, integrato nel IDE Eclipse. Il report è riportato a fine capitolo.

3.5.2 Analisi dinamica

Nell'iterazione 2 sono state testate tutte le API REST implementate, utilizzando Postman (Fig.3.4). In particolare sono state testate le seguenti funzionalità:

- UserController:
 - Login con credenziali corrette e verifica che le informazioni dell'utente ricevute siano corrette;
 - Login con credenziali errate;
 - Visualizzazione di tutti gli utenti registrati nel sistema;
 - Visualizzazione di un utente specifico e verifica corretta dei dati ritornati;
 - Eliminazione utente;
 - Inserimento di un nuovo utente e verifica che i dati dell'utente siano stati inseriti in modo corretto;
 - Modifica utente e verifica che l'utente sia stato modificato correttamente;
- TeamController:
 - Visualizzazione di una squadra e verifica che le informazioni ricevute siano corrette;
 - Visualizzazione di tutte le squadre inserite nel sistema;
 - Eliminazione di una squadra;
 - Inserimento di una nuova squadra;
 - Assegnamento di un caposquadra a una squadra;
- AreaController:
 - Visualizzazione di tutte le aree inserite nel sistema;

GET	Login corretto	localhost:8080/login	/ Login corretto	202 Accepted	616 ms	580 B
			Pass Codice di stato: 202; Utente autorizzato			
			Pass Utente ricevuto corretto			
GET	Login errato	localhost:8080/login	/ Login errato	401 Unauthorized	44 ms	393 B
			Pass Codice di stato: 401; Utente non autorizzato			
GET	Get all users	localhost:8080/users	/ Get all users	200 OK	382 ms	2.714 KB
			Pass Codice di stato: 200; Utenti ricevuti			
GET	Get one user	localhost:8080/users/1	/ Get one user	200 OK	252 ms	574 B
			Pass Codice di stato: 200, Utente ricevuto			
			Pass Utente ricevuto corretto			
DELETE	Delete one user	localhost:8080/users/2	/ Delete one user	202 Accepted	474 ms	311 B
			Pass Codice di stato: 202; Utente eliminato			
POST	Add a new user	localhost:8080/users	/ Add a new user	201 Created	206 ms	584 B
			Pass Codice di stato: 201; Utente creato			
			Pass Utente creato correttamente			
PUT	Modify a user	localhost:8080/users	/ Modify a user	200 OK	200 ms	572 B
			Pass Codice di stato: 200; Utente modificato			
			Pass Utente modificato correttamente			
GET	Get one team	localhost:8080/teams/1	/ Get one team	200 OK	253 ms	1.208 KB
			Pass Codice di stato: 200; Team ricevuto			
			Pass Team ricevuto corretto			
GET	Get all teams	localhost:8080/teams	/ Get all teams	200 OK	380 ms	2.245 KB
			Pass Codice di stato: 200; Squadre ricevute			
DELETE	Delete a team	localhost:8080/teams/3	/ Delete a team	202 Accepted	527 ms	311 B
			Pass Codice di stato: 202; Team eliminato			
POST	Create a team	localhost:8080/teams	/ Create a team	201 Created	213 ms	377 B
			Pass Codice di stato: 201; Squadra creata			
GET	Set id foraman	localhost:8080/teams/setforeman?idTeam=3&idForeman=6	/ Set id foraman	202 Accepted	315 ms	387 B
			Pass Codice di stato: 202; Caposquadra assegnato correttamente			
GET	Get all areas	localhost:8080/areas	/ Get all areas	200 OK	236 ms	737 B
			Pass Codice di stato: 200; Zone ricevute			

Figura 3.4: Risultati test API su Postman.

3.5.3 Unit Test

LvsEmergency Server

In questa iterazione è stata testata la funzione `findByUsername(String username)` che permette di recuperare l'utente inserito nel database con lo *username* specificato. Inoltre viene eseguito il test di avvio del framework Spring, inserito di default. Di seguito è riportato il codice del test.

```
1 package it.lvsemergency.accountManagement;
2
3 import static org.assertj.core.api.Assertions.assertThat;
4
5 import org.junit.jupiter.api.Test;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
8 import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
9
10 @DataJpaTest
11 @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
12 public class UserRepositoryTest {
13
14     @Autowired
15     private UserRepository underTest;
16
17     @Test
18     void findByUsername() {
19
20         String username = "test";
21         //given
22         User expected = new User(1, username, "Mario", "Rossi",
23             "MRARSS80A01C800V", "test", "Via Nazionale 7", "3485556255", 'M',
24             "mario@rossi.it", null, UserRole.ADMINISTRATOR, OperativityRole.ACTIVE);
25
26         underTest.save(expected);
27
28         //when
29         User result = underTest.findByUsername(username).get();
30
31         //then
32         assertThat(expected).isEqualTo(result);
33     }
34 }
```

Il risultato del test con JUnit ha confermato il corretto funzionamento della funzione.

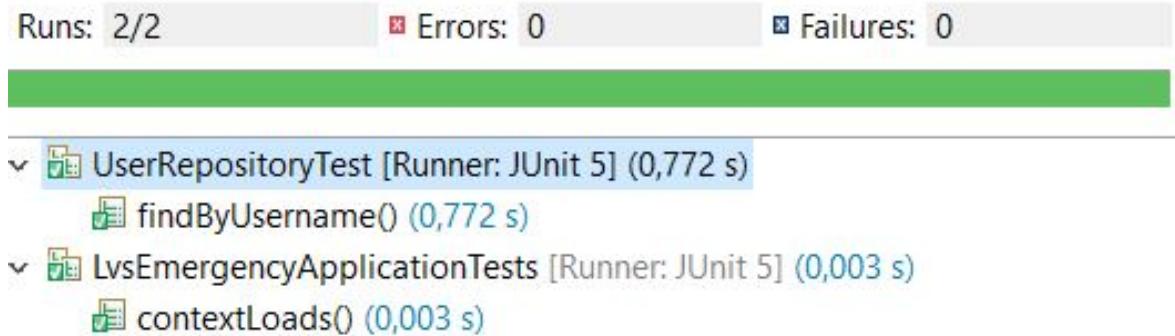


Figura 3.5: Risultato test con JUnit.

Client app

Lato client è stata testata la corretta creazione delle classi `User`, `Team` e `Area` a partire da una stringa JSON. Per fare ciò, è stato utilizzato il framework *Qt Test*. Per effettuare il parsing è stata utilizzata la libreria *QJsonDocument*, fornita da Qt: si tratta di una classe che è in grado di analizzare un documento JSON con funzionalità di lettura e scrittura. Di seguito viene riportato il codice del test.

```
1 #include "testentityif.h"
2
3
4 void TestEntityIF::testUserFromJson()
5 {
6     QString jsonString = "{\"idUser\": 11, \"username\": \"d.salvetti1\", "
7         "\"name\": \"Davide\", \"surname\": \"Salvetti\", "
8         "\"address\": \"Via Nazionale 21\", \"cellNumber\": "
9         "\"3470000000\", "
10        "\"sex\": \"M\", \"email\": \"d.salvetti1@studenti.unibg.it\" , "
11        "\"idTeam\": 3, \"role\": \"FOREMAN\", \"state\": \"ACTIVE\" , "
12        "
13        \"cf\": \"SLVDVD97A00A000A\" }";
14
15
16     accountmanagementIF::User *user = new accountmanagementIF::User();
17     user->fromJsonObject(QJsonDocument::fromJson(jsonString.toUtf8()).object());
18
19     QVERIFY(user->getIdUser() == 11);
20     QVERIFY(user->getUsername() == "d.salvetti1");
21     QVERIFY(user->getName() == "Davide");
22     QVERIFY(user->getAddress() == "Via Nazionale 21");
23     QVERIFY(user->getCellnumber() == "3470000000");
24     QVERIFY(user->getCf() == "SLVDVD97A00A000A");
25     QVERIFY(user->getRole() == 1);
26     QVERIFY(user->getSurname() == "Salvetti");
27     QVERIFY(user->getSex() == "M");
28     QVERIFY(user->getState() == 1);
29     QVERIFY(user->getTeam() == 3);
30 }
31
32 void TestEntityIF::testTeamFromJson()
33 {
34     QString jsonString = "{\"idTeam\":1, \"teamName\":\"BGTeam\", \"idForeman\":4, "
35         "\"users\":[{\"idUser\":1, \"username\": \"admin\", \"name\": \""
36         \"Mario\", "
37         "\"surname\": \"Rossi\", \"address\": \"Via Nazionale 7\", \""
38         \"cellNumber\": "
39         "\"3485556255\", \"sex\": \"M\", \"email\": \"mario@rossi.it\", "
40         "\"idTeam\":1, \"role\": \"ADMINISTRATOR\", \"state\": \"ACTIVE "
41         "\", \"cf\": \"MRARSS80A01C800V\", {"
42             "\"idUser\":2, \"username\": \"admin1\", \"name\": \"Daniela\", "
43             "\"surname\": \"Rossi\", \"address\": "
44             "\"Via Nazionale 8\", \"cellNumber\": \"325186255\", \"sex\": \"F "
45             "\", \"email\": \"daniela@rossi.it\", "
46             "\"idTeam\":1, \"role\": \"ADMINISTRATOR\", \"state\": \"ACTIVE "
47             "\", \"cf\": \"DNARSS80A01C800V\"}"
48         "], \"area\":[{\"idArea\":1, \"areaName\": \"Bergamo\", \"lat "
49         \"\":45.69499969482422, \"lng\": 9.670000076293945, \"istatCode\": \"016024\"}] }";
50
51     teamManagementIF::Team *team = new teamManagementIF::Team();
52     team->fromJsonObject(QJsonDocument::fromJson(jsonString.toUtf8()).object());
53
54     QVERIFY(team->getTeamName() == "BGTeam");
55     QVERIFY(team->getIdTeam() == 1);
56     QVERIFY(team->getIdForeman() == 4);
57
58     areaInformationManagementIF::Area *area = team->getArea();
```

```

49
50     QVERIFY(area->getAreaName() == "Bergamo");
51     QVERIFY(area->getIstatCode() == "016024");
52     QVERIFY(area->getIdArea() == 1);
53     QVERIFY(area->getLat() == 45.69499969482422);
54     QVERIFY(area->getLng() == 9.670000076293945);
55
56     QList<accountmanagementIF::User *> users = team->getUsers();
57
58     QVERIFY(users.at(0)->getUsername() == "admin");
59     QVERIFY(users.at(0)->getName() == "Mario");
60     QVERIFY(users.at(0)->getSurname() == "Rossi");
61
62     QVERIFY(users.at(1)->getUsername() == "admin1");
63     QVERIFY(users.at(1)->getName() == "Daniela");
64     QVERIFY(users.at(1)->getSurname() == "Rossi");
65 }
66
67 void TestEntityIF::testAreaFromJson()
68 {
69     QString jsonString = "{\"idArea\":3,\"areaName\":\"Costa di Mezzate\",\"lat\":
70         \"45.66666793823242,\"
71         \"lng\":9.800000190734863,\"istatCode\":\"016084\"}";
72
73     areaInformationManagementIF::Area *area = new areaInformationManagementIF::Area();
74     area->fromJsonObject(QJsonDocument::fromJson(jsonString.toUtf8()).object());
75
76     QVERIFY(area->getAreaName() == "Costa di Mezzate");
77     QVERIFY(area->getIstatCode() == "016084");
78     QVERIFY(area->getIdArea() == 3);
79     QVERIFY(area->getLat() == 45.66666793823242);
80     QVERIFY(area->getLng() == 9.800000190734863);
81
82
83 QTest_MAIN(TestEntityIF)

```

Il risultato del test ha confermato il corretto funzionamento delle funzioni.

```

PacClientAppTest X
Config: Using QTest library 5.15.2, Qt 5.15.2 (x86_64-little_endian-llp64 shared (dynamic) release build; by GCC 8.1.0), windows 10
PASS : TestEntityIF::initTestCase()
PASS : TestEntityIF::testUserFromJson()
PASS : TestEntityIF::testTeamFromJson()
PASS : TestEntityIF::testAreaFromJson()
PASS : TestEntityIF::cleanupTestCase()
Totals: 5 passed, 0 failed, 0 skipped, 0 blacklisted, 1ms
***** Finished testing of TestEntityIF *****
09:00:25: C:\Users\David\Desktop\Progetto_PAC\ClientApp\build-PacClientApp-Desktop_Qt_5_15_2_MinGW_64_bit-Debug\test\debug\PacClientAppTest.exe exited with code 0

```

Figura 3.6: Risultato test con Qt Test.

3.6 Documentazione API

In questa sezione viene mostrata la documentazione relativa ad alcune API implementate nell'iterazione 2. È possibile visualizzare una collezione (creata con Postman) di tutte le API realizzate e testate sulla repository GitHub. In particolare, si riportano di seguito le più significative:

- API per il login (nel header della richiesta sono state inserite correttamente le informazioni secondo il protocollo della Basic Authentication);
- API per la visualizzazione di un utente specifico, il cui id è inserito nel path della chiamata;

- API per la modifica di un utente, inserendo nel body della richiesta l'utente modificato;
- API per visualizzare un team, il cui id viene specificato nel path della richiesta;
- API per la creazione di un team, inserendo nel body della richiesta le informazioni del team;
- API per l'assegnamento di un caposquadra a una squadra;
- API per la visualizzazione di tutte le area inserite nel sistema;

GET Login corretto [Open Request →](#)

localhost:8080/login

```

1 {
2   "idUser": 1,
3   "username": "admin",
4   "name": "Mario",
5   "surname": "Rossi",
6   "address": "Via Nazionale 7",
7   "cellNumber": "3485556255",
8   "sex": "M",
9   "email": "mario@rossi.it",
10  "idTeam": null,
11  "role": "ADMINISTRATOR",
12  "state": "ACTIVE",
13  "cf": "MRARSS80A01C800V"
14 }
```

Figura 3.7: Documentazione API Login.

GET Get one user [Open Request →](#)

localhost:8080/users/4

```

1 {
2   "idUser": 4,
3   "username": "foreman1",
4   "name": "Giancarlo",
5   "surname": "Bianchi",
6   "address": "Via Verdi 1",
7   "cellNumber": "3215186255",
8   "sex": "M",
9   "email": "giancarlo@bianchi.it",
10  "idTeam": 1,
11  "role": "FOREMAN",
12  "state": "ACTIVE",
13  "cf": "BNARSS80A01C800V"
14 }
```

Figura 3.8: Documentazione API Get one user.

PUT Modify a user

[Open Request →](#)

localhost:8080/users

Body:

```
1 {  
2   "idUser": 3,  
3   "username": "admin2",  
4   "password": "admin2",  
5   "name": "Luca",  
6   "surname": "Rossi",  
7   "address": "Via Nazionale 9",  
8   "cellNumber": "3252186255",  
9   "sex": "M",  
10  "email": "luca@rossi.it",  
11  "idTeam": null,  
12  "role": "ADMINISTRATOR",  
13  "state": "ACTIVE",  
14  "cf": "ABCSS80A01C800V"  
15 }
```

Response:

```
1 {  
2   "idUser": 3,  
3   "username": "admin2",  
4   "password": "admin2",  
5   "name": "Luca",  
6   "surname": "Rossi",  
7   "address": "Via Nazionale 9",  
8   "cellNumber": "3252186255",  
9   "sex": "M",  
10  "email": "luca@rossi.it",  
11  "idTeam": null,  
12  "role": "ADMINISTRATOR",  
13  "state": "ACTIVE",  
14  "cf": "ABCSS80A01C800V"  
15 }
```

Figura 3.9: Documentazione API Modify a user.

GET Get one team

[Open Request →](#)

localhost:8080/teams/1

```
1  {
2      "idTeam": 1,
3      "teamName": "BGTeam",
4      "idForeman": 4,
5      "users": [
6          {
7              "idUser": 1,
8              "username": "admin",
9              "name": "Mario",
10             "surname": "Rossi",
11             "address": "Via Nazionale 7",
12             "cellNumber": "3485556255",
13             "sex": "M",
14             "email": "mario@rossi.it",
15             "idTeam": 1,
16             "role": "VOLUNTEER",
17             "state": "ACTIVE",
18             "cf": "MRARSS80A01C800V"
19         },
20         {
21             "idUser": 4,
22             "username": "foreman1",
23             "name": "Giancarlo",
24             "surname": "Bianchi",
25             "address": "Via Verdi 1",
26             "cellNumber": "3215186255",
27             "sex": "M",
28             "email": "giancarlo@bianchi.it",
29             "idTeam": 1,
30             "role": "FOREMAN",
31             "state": "ACTIVE",
32             "cf": "BNARSS80A01C800V"
33         }
34     ],
35     "area": {
36         "idArea": 1,
37         "areaName": "Bergamo",
38         "lat": 45.69499969482422,
39         "lng": 9.670000076293945,
40         "istatCode": "016024"
41     }
42 }
```

Figura 3.10: Documentazione API Get one team.

POST Create a team [Open Request →](#)

localhost:8080/teams

Body:

```
1 {
2   "teamName": "LVSTeam",
3   "idArea": "4",
4   "idForeman": "3"
5 }
```

Response:
Team created successfully!

Figura 3.11: Documentazione API Create a team.

GET Set id foraman [Open Request →](#)

localhost:8080/teams/setforeman?idTeam=3&idForeman=6

Response:
Foreman 6 assigned to 3 correctly !

Figura 3.12: Documentazione API Set id foreman.

GET Get all areas

[Open Request →](#)

localhost:8080/areas

```
1 [  
2   {  
3     "idArea": 1,  
4     "areaName": "Bergamo",  
5     "lat": 45.69499969482422,  
6     "lng": 9.670000076293945,  
7     "istatCode": "016024"  
8   },  
9   {  
10    "idArea": 2,  
11    "areaName": "Sovere",  
12    "lat": 45.81666564941406,  
13    "lng": 10.033333778381348,  
14    "istatCode": "016204"  
15  },  
16  {  
17    "idArea": 3,  
18    "areaName": "Costa di Mezzate",  
19    "lat": 45.66666793823242,  
20    "lng": 9.800000190734863,  
21    "istatCode": "016084"  
22  },  
23  {  
24    "idArea": 4,  
25    "areaName": "Dalmine",  
26    "lat": 45.650001525878906,  
27    "lng": 9.600000381469727,  
28    "istatCode": "016091"  
29  }  
30 ]
```

Figura 3.13: Documentazione API Get all areas.

Quality Report

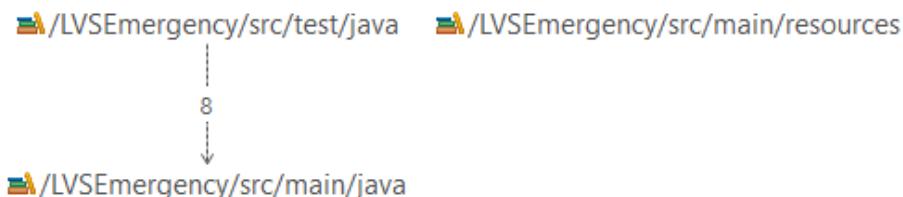
Creation Date 2022-02-20

Package Prefix it.

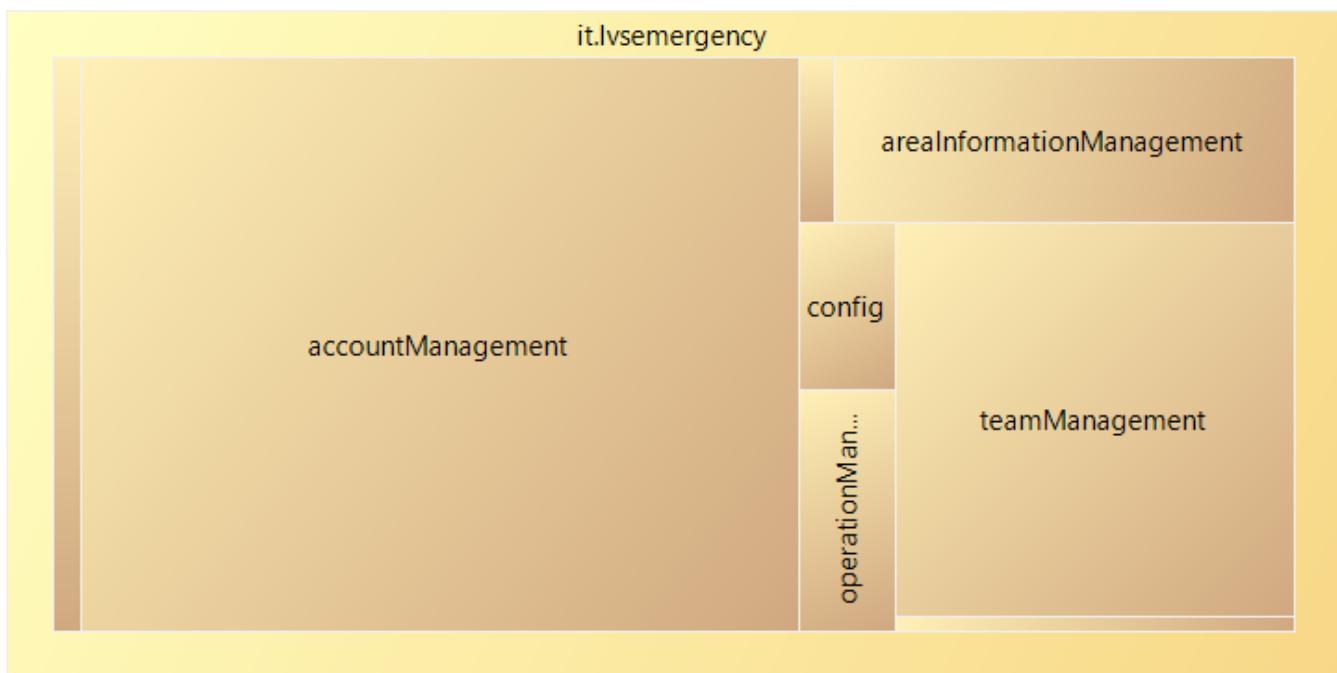
Level of Detail Member

Tmp_220123_212257

Library Dependency Graph



Treemap Overview



Metrics Summary

Metric	Value
Number of Libraries	3
Number of Packages	8
Number of Top Level Classes	26
Average Number of Top Level Classes per Package	3.25
Average Number of Member Classes per Class	0
Average Number of Methods per Class	7.15
Average Number of Fields per Class	2.23
Estimated Lines of Code	760
Estimated Lines of Code per Top Level Class	29.23

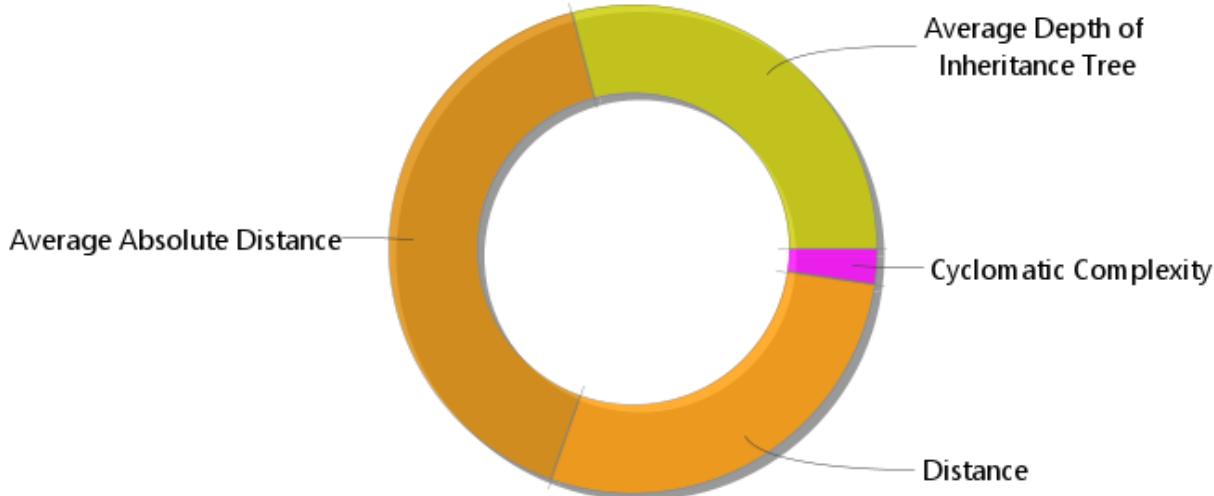
Average Cyclomatic Complexity	0.95
Fat for Library Dependencies	1
Fat for Flat Package Dependencies	3
Fat for Top Level Class Dependencies	47
Tangled for Library Dependencies	0%
Average Component Dependency between Libraries	16.67%
Average Component Dependency between Packages	5.36%
Average Component Dependency between Units	13.69%
Average Distance	0.12
Average Absolute Distance	0.47
Average Weighted Methods per Class	6.81
Average Depth of Inheritance Tree	0.81
Average Number of Children	0
Average Coupling between Objects	0.92
Average Response for a Class	8.69
Average Lack of Cohesion in Methods	27.54

Top Violations (7 of 7)

Artifact	Metric	Value
Tmp_220123_212257	D	0.47
Tmp_220123_212257	DIT	0.81
lvsemergency.accountManagement	D	-0.80
lvsemergency.areaInformationManagement	D	-0.60
lvsemergency.accountManagement.User.equals(...)	CC	17
lvsemergency.operationManagement	D	1
lvsemergency.alarmNotificationManagement	D	1

Pollution Chart

Pollution 1.37



Violations by Metric

Cyclomatic Complexity

Artifact	Value
lvsemergency.accountManagement.User.equals(...)	17

Distance

Artifact	Value
lvsemergency.accountManagement	-0.80
lvsemergency.areaInformationManagement	-0.60
lvsemergency.operationManagement	1
lvsemergency.alarmNotificationManagement	1

Average Absolute Distance

Artifact	Value
Tmp_220123_212257	0.47

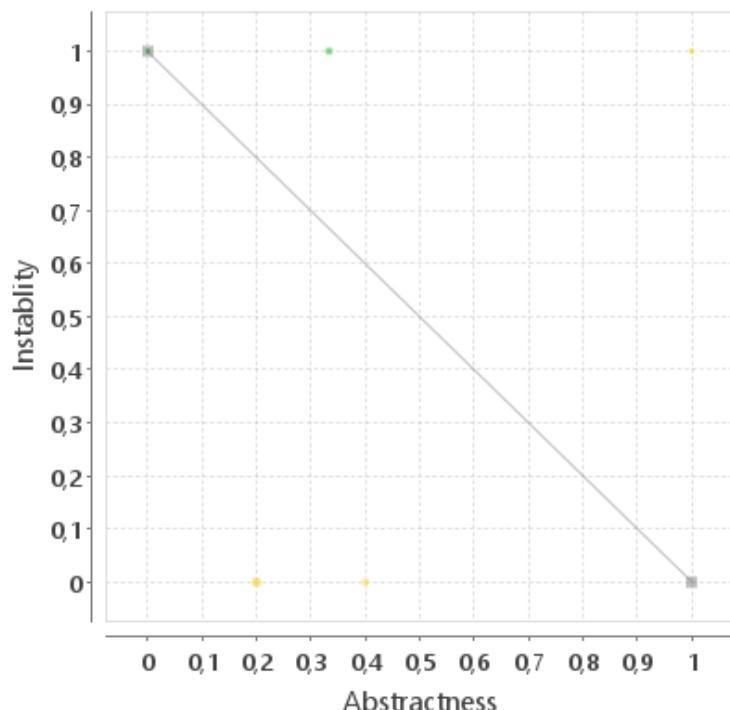
Average Depth of Inheritance Tree

Artifact	Value
Tmp_220123_212257	0.81

Design Tangles

There are no design tangles.

Package Distance Chart



Metric Ratings

Count Metrics

Metric	Rating	Linear
Number of Top Level Classes	20 40 60 80	✓
Number of Methods	25 50 100 200	✓
Number of Fields	10 20 40 80	✓
Estimated Lines of Code	200 300 400 500	✓
Estimated Lines of Code	30 60 120 240	✓

Complexity Metrics

Metric	Rating	Linear
Cyclomatic Complexity	10 15 20 30	✓
Fat	30 60 120 240	✓
Fat	30 60 120 240	✓
Fat	30 60 120 240	✓
Tangled	0 1	✓
Tangled for Library Dependencies	0 1	✓
Average Component Dependency between Libraries	0 5 1	✓
Average Component Dependency between Packages	0 5 1	✓

Robert C. Martin Metrics

Metric	Rating	Linear
Distance	-1 -5 0 5 1	✓
Average Absolute Distance	0 4 5 1	✓

Chidamber & Kemerer Metrics

Metric	Rating	Linear
Weighted Methods per Class	0 100 200 300	✓
Depth of Inheritance Tree	4 6 8 10	✓
Average Depth of Inheritance Tree	2 1 0	✓
Coupling between Objects	0 25 250	✓
Response for a Class	0 100 1000	✓

Capitolo 4

Iterazione 3

4.1 Introduzione

Nella terza iterazione sono stati implementati i componenti *Data Analyzer* e *Data Collector*, che permettono di raccogliere e analizzare i dati di interesse per la nostra applicazione. Inoltre, sono state implementate le API per la visualizzazione dei dati ambientali raccolti e degli allarmi generati. Infine, è stato implementato il caso d'uso "Visualizzazione posizione real-time" che utilizza delle API di *Open Street Map* grazie alle quali è possibile mostrare la posizione degli operatori su una mappa. L'accesso alle API di *Open Street Map* è semplificato dal modulo *Qt Location* fornito da Qt.

Grazie all'implementazione di questi due componenti, è stato possibile realizzare anche i seguenti casi d'uso:

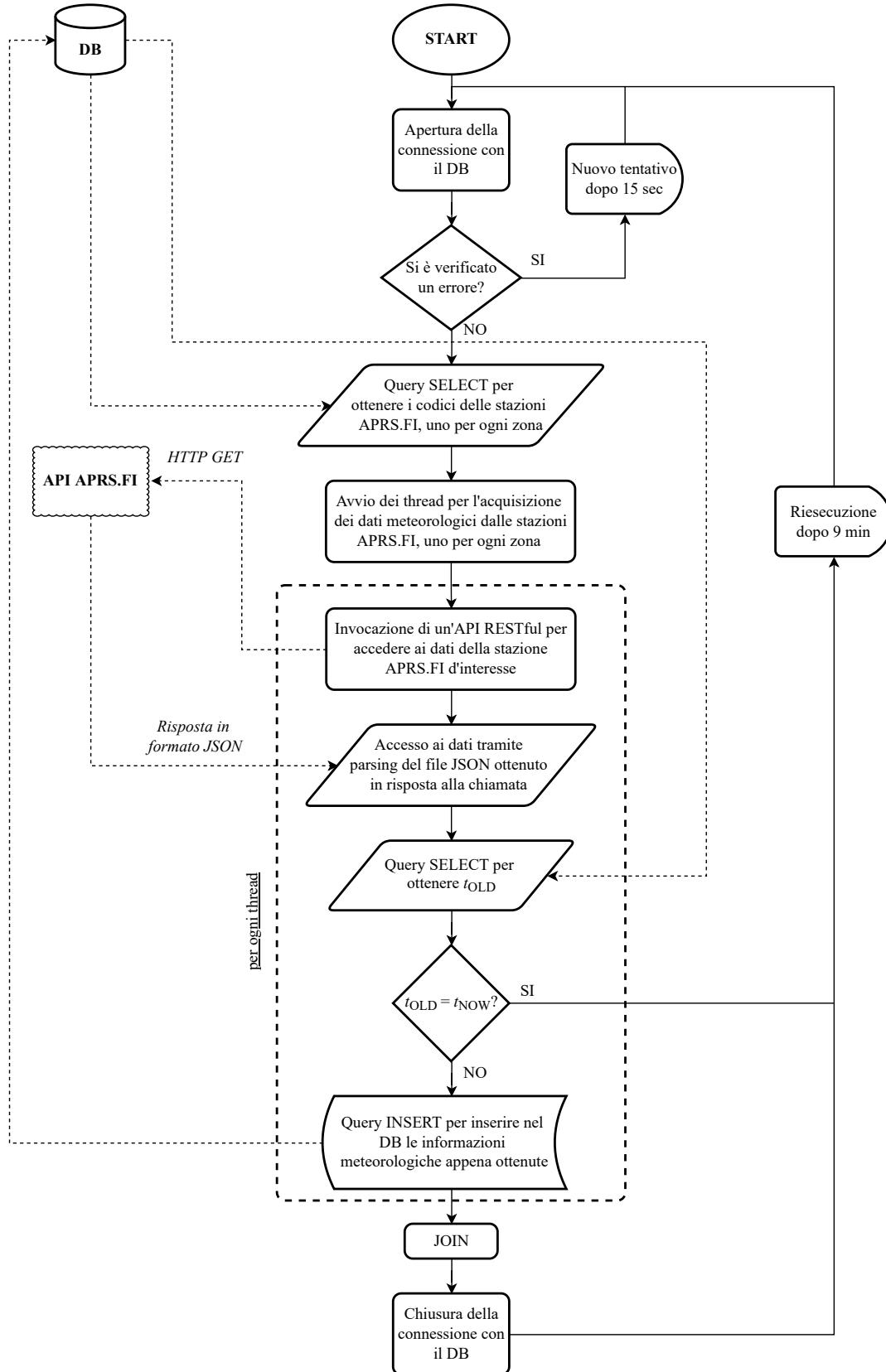
- UC18 - Visualizzazione informazioni zona [astratto];
 - UC18.1 - Visualizzazione dati ambientali;
 - UC18.1 - Visualizzazione allarmi;
- UC10 - Visualizzazione posizione real-time

4.1.1 Data Collector

Il *Data Collector* è un componente fondamentale dell'applicazione: il suo compito è quello di raccogliere i dati dai data server tramite API. In particolare, in questa iterazione è stata implementata la raccolta dei dati meteorologici tramite l'invocazione dell'API messa a disposizione dal sito APRS.FI (<https://aprs.fi/>) e descritta nel paragrafo 4.1.1.

Si è scelto di implementare questo componente come una applicazione scritta in Python, che verrà poi distribuita su un Raspberry Pi 4, in modo da garantire l'esecuzione continua. Per descriverne il funzionamento, si è costruito il diagramma di flusso rappresentato di seguito (Fig.4.1). Inizialmente, l'applicazione tenta la creazione di una connessione con il database. Se si verificano errori, il sistema attenderà 15 secondi prima di effettuare un nuovo tentativo di connessione. In questo modo, se si dovessero presentare problemi, come mancanza di connessione internet o irraggiungibilità del DB, il sistema effettuerà un nuovo tentativo di connessione. Una volta stabilita una connessione con il DB, vengono recuperati tutti i codici identificativi delle stazioni meteo (al massimo uno per ogni zona). Successivamente, viene avviato un thread per ogni zona, che si occuperà di recuperare i dati meteorologici dall'API, attraverso una chiamata HTTP di tipo GET. Una volta ricevuta la risposta in formato JSON, viene confrontata la data dell'acquisizione ricevuta con quella dell'ultima misurazione disponibile già inserita. Se essa coincide, il thread termina la sua esecuzione, altrimenti procede all'inserimento nel DB della nuova acquisizione. Questo controllo è stato inserito poiché le stazioni non hanno frequenze di aggiornamento uguali e costanti. In questo modo, si evita di memorizzare

più volte la medesima acquisizione. Dopo aver salvato le informazioni nel database, viene attesa la terminazione di tutti i thread avviati e viene chiusa la connessione con il DB. Il programma attende 9 minuti prima di ricominciare il ciclo.



43
Figura 4.1: diagramma di flusso relativo all'esecuzione del componente Data Collector.

API APRS.FI Nella figura Fig.4.2 è riportato un esempio di chiamata all'API fornita da APRS.FI e la successiva risposta ricevuta. Affinché si possano ricevere i dati relativi ad una stazione è necessario definire i seguenti parametri:

- *name*: identificativo della stazione meteorologica;
- *what*: definizione della tipologia della stazione. Nel nostro caso siamo interessati a stazioni meteorologiche identificate dal valore *wx*;
- *apikey*: API key identificativo dell'utente che richiede i dati;
- *format*: formato della risposta ricevuta (può essere scelto JSON o XML);

La risposta conterrà, se la richiesta è andata a buon fine, i seguenti campi:

- *command*: tipo di richiesta http;
- *result*: indica se la richiesta è andata a buon fine;
- *found*: numero di stazione trovate;
- *name*: identificativo della richiesta;
- *time*: istante di acquisizione della misurazione corrente, nel formato *Unix time*;
- *temp*: temperatura in gradi Celsius;
- *pressure*: pressione in mbar;
- *humidity*: umidità relativa;
- *wind_direction*: direzione del vento in gradi;
- *wind_speed*: velocità del vento media in metri al secondo;
- *wind_gust*: raffiche di vento in metri al secondo;
- *rain_1h*: pioggia cumulata nell'ultima ora in millimetri;
- *rain_24h*: pioggia cumulata nell'ultimo giorno in millimetri;
- *rain_mn*: pioggia cumulata dalla mezzanotte in millimetri;
- *luminosity*: luminosità rilevata;

Tutti i campi sono opzionali e dipendono dai sensori attivi sulla particolare stazione. Tutta la documentazione sulle API esposte dal sito APRS.FI è disponibile all'indirizzo <https://aprs.fi/page/api>.



The screenshot shows a browser interface with a GET request to <https://api.aprs.fi/api/get?name=FW0796&what=wx&apikey=165780.vwUJxymaP4yljUx&format=json>. The response is a JSON object with the following structure:

```
1 {  
2   "command": "get",  
3   "result": "ok",  
4   "found": 1,  
5   "what": "wx",  
6   "entries": [  
7     {  
8       "name": "FW0796",  
9       "time": "1643649147",  
10      "temp": "10.6",  
11      "pressure": "1006.0",  
12      "humidity": "30",  
13      "wind_direction": "315",  
14      "wind_speed": "1.8",  
15      "wind_gust": "2.7",  
16      "rain_1h": "0.0",  
17      "rain_24h": "0.0",  
18      "rain_mn": "0.0"  
19    }  
20  ]  
21}
```

Figura 4.2: Esempio chiamata API APRS.FI

4.1.2 Data Analyzer

Il *Data Analyzer* si occupa dell’analisi dei dati raccolti dal *Data Collector* al fine di generare gli allarmi che saranno poi visibili agli utenti tramite l’applicazione client. Come per il *Data Collector*, si è deciso di implementare questo componente come un applicativo Python che verrà in seguito distribuito su un Raspberry Pi4 per la sua esecuzione periodica. Si è scelto di utilizzare questo linguaggio di programmazione poiché sono disponibili numerose librerie per effettuare analisi dei dati, moduli e package che verranno sfruttati per la generazione delle allerte. In particolare, si è scelto di implementare queste ultime per segnalare i rischi nebbia o brina e l’imminente arrivo di una forte perturbazione. Nella figura 4.3 è rappresentato il diagramma di flusso relativo al Data Analyzer. All’avvio dell’applicazione vengono inizializzate alcune strutture dati di supporto all’esecuzione ciclica degli algoritmi per l’analisi dei dati, algoritmi che sono descritti dettagliatamente in seguito. Successivamente, viene eseguito un tentativo di connessione con il DB seguendo una strategia uguale a quella utilizzata per il *Data Collector*. Se la connessione viene stabilita, allora vengono recuperati dal DB e memorizzati i codici identificativi delle stazioni meteorologiche; essi saranno utilizzati successivamente per accedere ai dati da fornire in input agli algoritmi per la generazione degli allarmi. Dopodiché, viene avviato un thread per ogni zona, il quale si occuperà della generazione delle allerte maltempo tramite l’analisi dei dati relativi alla pressione atmosferica. Altresì vengono avviati i thread che si occupano di analizzare i dati di temperatura e umidità relativa al fine di creare le allerte nebbia e brina. Infine, si attende la terminazione di tutti i thread precedentemente avviati e, a join concluso, viene chiusa la connessione con il DB. Da notare che la periodicità del ciclo è di 5 minuti.

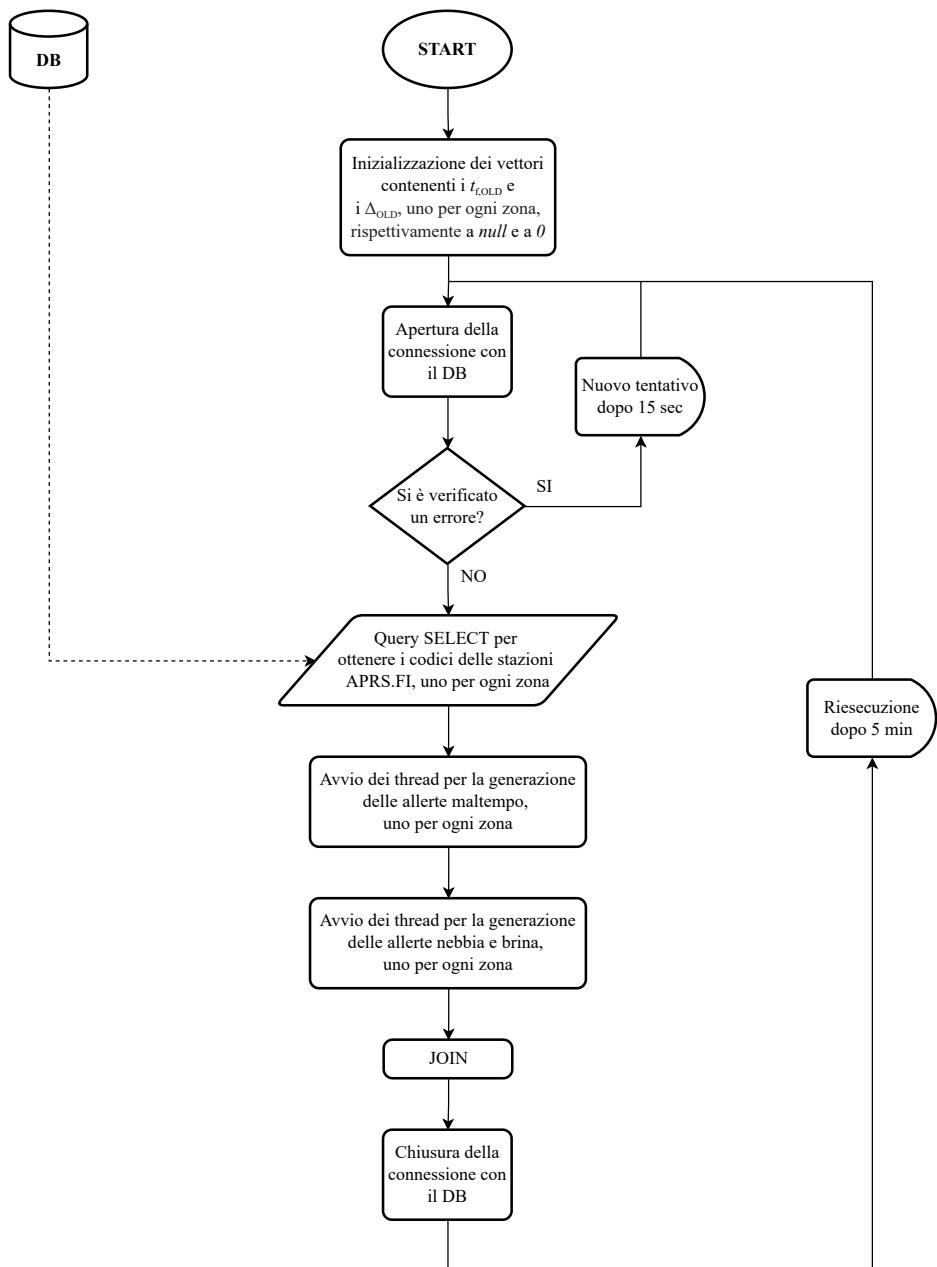


Figura 4.3: diagramma di flusso relativo all'esecuzione del componente Data Analyzer.

Generatore di allerte maltempo

Cenni di meteorologia Esistono due differenti approcci per prevedere l'andamento delle condizioni meteorologiche a partire dalle rilevazioni della pressione atmosferica p_{atm} :

- *generale* (analisi della p_{atm} in termini assoluti): se la pressione è inferiore a 1000 hPa, allora è probabile che il tempo volga brutto; tale probabilità aumenta in presenza di venti meridionali e di un'umidità superiore al 50 %. Se, invece, la pressione è superiore a 1025 hPa, allora è probabile che il tempo tenda al bello; tale probabilità aumenta in presenza di venti settentrionali e di un'umidità inferiore al 60 %;
- *specifico* (analisi della p_{atm} in termini di variazioni temporali): un calo di 1-2 hPa in 3 ore precorre un peggioramento che si manifesta entro le prossime 24-48 ore, una diminuzione superiore a 2-3 hPa in 3 ore entro le prossime 12-24 ore, mentre un calo di 5-6 hPa in 3 ore sta a indicare un peggioramento imminente per di più da associare a una perturbazione violenta.

La pressione atmosferica, inoltre, è soggetta a un andamento periodico nel corso del giorno:

- primo minimo alle ore 4;
- primo massimo alle ore 10;
- secondo minimo alle ore 16;
- secondo massimo alle ore 22.

Pertanto, è importante innanzitutto destagionalizzare la serie storica affinché possano essere studiate le variazioni relative esclusivamente a un cambiamento delle condizioni meteorologiche.

Descrizione dell'algoritmo Il primo passo consiste nell'accedere al DB per ottenere la data e l'ora dell'acquisizione più recente di pressione atmosferica, ossia t_f , con riferimento alla stazione APRS.FI a cui è associata la zona d'interesse su cui il thread sta operando. Se t_f risulta essere uguale a $t_{f,OLD}$, ossia all'istante temporale rispetto al quale è stata generata l'allerta più recente, allora significa che il Data Collector non ha inserito nel DB una nuova rilevazione durante i 5 min di attesa, quindi l'algoritmo termina perché sarebbe inutile creare un'allerta basata su dei dati che non sono ancora stati aggiornati; il fine è evitare ripetizioni. Una volta verificata la diversità tra t_f e $t_{f,OLD}$, si accede alla serie storica della pressione atmosferica delle ultime 24 ore e la si destagionalizza; p_{atm} , come anticipato in precedenza, è caratterizzata, infatti, da un andamento giornaliero che dev'essere eliminato affinché possano essere osservate le variazioni associate esclusivamente a un cambiamento delle condizioni meteorologiche. Per rimuovere questa componente di non-stazionarietà è sufficiente sottrarre a ogni campione la media in orizzontale. Dopodiché, vengono definiti i seguenti intervalli:

- $I_1 = \{\text{sottoinsieme dei dati di } p_{atm}(t) \text{ contenente gli istanti temporali relativi alle prime 2 rilevazioni di a partire dall'istante 3 ore precedente a } t_f\}$
- $I_2 = \{\text{sottoinsieme dei dati di } p_{atm}(t) \text{ contenente gli istanti temporali relativi alle ultime 2 rilevazioni}\}.$

Questi ultimi vengono a loro volta utilizzati per calcolare i seguenti parametri, valori necessari per implementare l'approccio *specifico*, ossia il più affidabile, per prevedere l'andamento delle condizioni meteorologiche:

$$\bullet \bar{p}_{low} = \frac{1}{2} \cdot \sum_{i=1}^{I_1} p_{atm}(i);$$

$$\bullet \bar{p}_{up} = \frac{1}{2} \cdot \sum_{i=1}^{I_2} p_{atm}(i).$$

$$\bullet \Delta = \bar{p}_{up} - \bar{p}_{low}$$

Si sarebbe potuto calcolare la variazione "pura" sottraendo al valore di pressione atmosferica più recente quello 3 ore precedente, ma si è preferito calcolare le medie \bar{p}_{low} e \bar{p}_{up} per limitare i disturbi dovuti a errori di misura dei sensori barometrici installati sulle stazioni APRS.FI. Da notare che Δ descrive la variazione *media* di p_{atm} nelle ultime 3 ore. In base al valore assunto da quest'ultimo parametro vengono create le seguenti allerte:

- se $\Delta \geq 0$, allora viene generata un'allerta di tipo *NONE*: la pressione atmosferica è in aumento, quindi non è previsto l'arrivo di una perturbazione;
- altrimenti se $\Delta > \Delta_{OLD}$, allora viene creata un'allerta di tipo *NONE*: il maltempo si sta allontanando oppure un picco negativo di p_{atm} è stato superato poiché la variazione è più piccola rispetto alla rilevazione precedente;
- altrimenti se $\Delta < 5 \text{ hPa}$, allora viene generata un'allerta di tipo *RED*: una violenta perturbazione è in avvicinamento ed è imminente;
- altrimenti viene creata un'allerta di tipo *NONE*: la pressione è in diminuzione, tuttavia la variazione nelle ultime 3 ore non è tale da destare allarmismi. Una perturbazione potrebbe essere in avvicinamento, tuttavia, se fosse violenta, non sopraggiungerebbe in tempi brevi.

Dopo aver inserito l'allerta nel DB tramite una query, i parametri $t_{f,OLD}$ e Δ_{OLD} vengono aggiornati rispettivamente con t_f e Δ in modo tale che i nuovi valori possano essere utilizzati come termini di confronto nell'iterazione successiva dell'algoritmo.

Da sottolineare, infine, il fatto che non sia stato utilizzato un modello statistico a memoria breve per prevedere l'andamento della pressione atmosferica nei minuti successivi in quanto una sua variazione, anche notevole, non preannuncia l'arrivo del maltempo in tempi stretti (meno di 1 ora), perciò le informazioni ottenibili studiando i dati delle ultime 3 ore sono sufficientemente esaustivi per fare previsione.

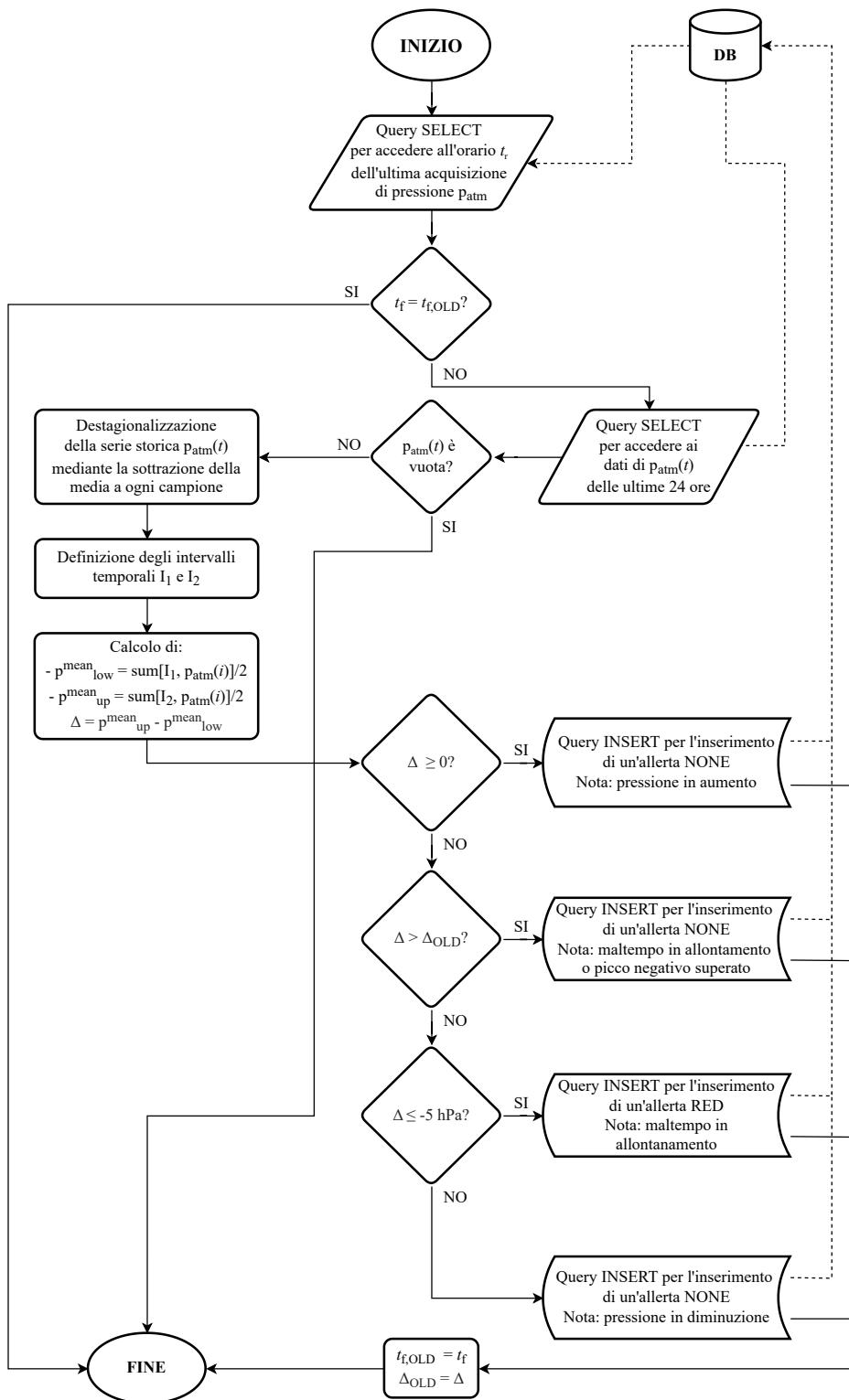


Figura 4.4: diagramma di flusso relativo all'algoritmo di generazione delle allerte maltempo.

Generatore di allerte nebbia e brina

Cenni di meteorologia Per prevedere le formazioni di nebbia e brina è necessario calcolare il *punto di rugiada* T_d , ossia la temperatura alla quale l'acqua contenuta nell'aria di un ambiente condensa e si trasforma in gocce d'acqua; esso viene raggiunto quando l'umidità relativa raggiunge il 100 %, cioè nell'istante in cui l'aria diventa satura e non riesce più a contenere l'umidità ambientale. Il punto di rugiada si determina tramite l'*approssimazione di Magnus-Tetens*:

$$T_d = \frac{b \cdot \alpha(T, UR)}{a - \alpha(T, UR)}$$

$$\text{con } \alpha(T, UR) = \frac{a \cdot T}{b + T} + \ln(UR), \quad a = 17,27 \text{ e } b = 237,7^\circ\text{C}$$

T (temperatura misurata): $-20^\circ\text{C} < T < 60^\circ\text{C}$

UR (umidità relativa): $0,01 < UR < 1,00$

La nebbia si forma quando $T = T_d$, mentre la brina quando $T = T_d$ con $T_d < 0$

Descrizione dell'algoritmo Per generare le allerte nebbia e brina si deve calcolare la temperatura di rugiada T_d (o *dew point*), la quale a sua volta è funzione della temperatura T e dell'umidità relativa UR . Pertanto, dopo aver verificato la diversità tra t_f e $t_{f,OLD}$, il primo passo dell'algoritmo (figura 4.5), viene eseguita una query per accedere ai dati di temperatura delle ultime 24 ore rilevati dalla stazione APRS.FI associata alla zona di competenza del thread. Questo parametro atmosferico dev'essere studiato tramite un modello statistico affinché possa essere previsto il suo comportamento nel breve termine; la scelta è ricaduta sull'ARIMA, un modello ARMA che rende stazionaria, se già non lo fosse, la serie storica tramite differenziazioni. Per identificare il modello, ossia per stimare il valore dei suoi parametri a massima verosimiglianza (o MLE), e per svolgere l'analisi di complessità, una procedura basata sulla minimizzazione dell'informazione di Akaike (o AIC), è stata utilizzata la funzione *auto_arima* del package *pmdarima* disponibile per il linguaggio Python. Da notare che a ogni iterazione viene stimato un nuovo modello utilizzando anche l'acquisizione più recente presente nel DB. Una volta conclusa l'operazione di stima dell'ARIMA, esso viene impiegato per svolgere una predizione a 3 passi della temperatura, ossia a:

- $t_1 = t_f + k;$
- $t_2 = t_f + 2 \cdot k;$
- $t_3 = t_f + 3 \cdot k;$

con k che rappresenta il tempo che intercorre tra due rilevazioni consecutive da parte della stazione meteorologica (circa 10 min) e t_f la data e l'ora dell'acquisizione più recente presente nel DB. Lo stesso studio a scopo predittivo viene fatto anche per l'umidità relativa. Una volta note le previsioni di T e UR , esse vengono combinate per fare la predizione a 3 passi della temperatura di rugiada T_d . Dopodiché, vengono calcolati gli intervalli di validità I, IC_1 , IC_2 e IC_3 entro i quali la temperatura deve rientrare affinché venga generata un'allerta. Essi sono necessari non solo perché mai si otterrà un'uguaglianza tra $T(t)$ e $T_d(t)$ a causa degli errori di predizione, ma anche perché la condensazione dell'acqua contenuta nell'aria può avvenire a una temperatura superiore a quella di rugiada; T_d , infatti, non tiene conto né della temperatura del suolo, il quale potrebbe essere più freddo della colonna d'aria che lo sovrasta, né della presenza di sostanze chimiche capaci di alterare il dew point. Gli intervalli sono asimmetrici poiché T non può essere più piccola di T_d ; se così fosse l'umidità relativa supererebbe il 100 %, un evento fisicamente impossibile. Per i criteri di generazione delle allerte si invita a visualizzare la figura 4.6; si noti che l'allerta brina viene creata quando la temperatura, attuale $T(t_f)$ o prevista $\hat{T}(t_1|t_f)$, $\hat{T}(t_2|t_f)$ e $\hat{T}(t_3|t_f)$:

- rientra nel proprio intervallo di validità (es. $\hat{T}(t_1|t_f) \in \text{IC}_1 = [\hat{T}_d(t_1|t_f); \hat{T}_d(t_1|t_f) + 0.45^\circ\text{C}]$);
- è non-positiva (es. $\hat{T}(t_1|t_f) \leq 0$).

L'algoritmo, infine, termina con l'aggiornamento di $t_{f,OLD}$.

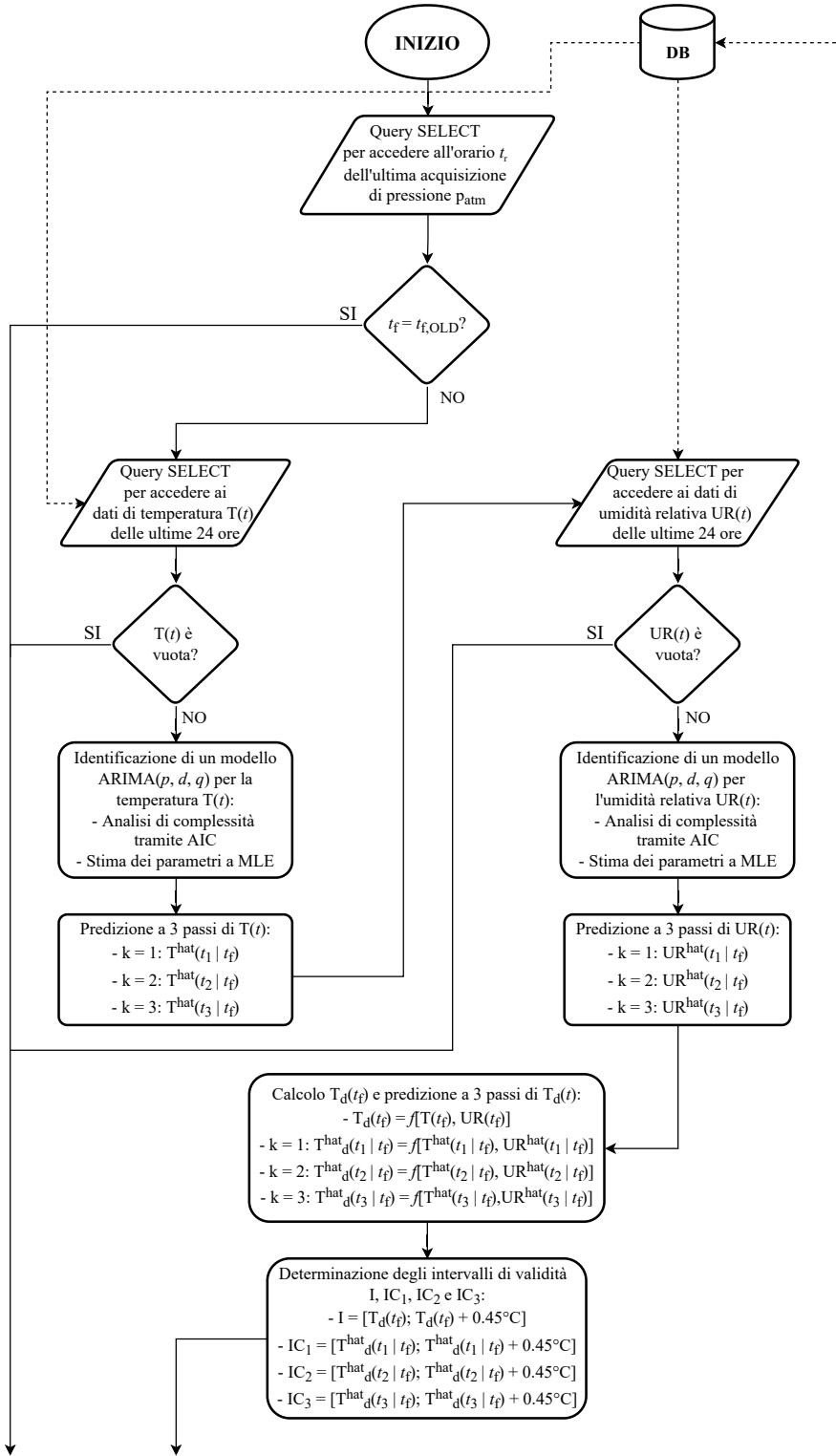


Figura 4.5: Diagramma di flusso relativo all'algoritmo di generazione delle allerte nebbia e brina, prima parte.

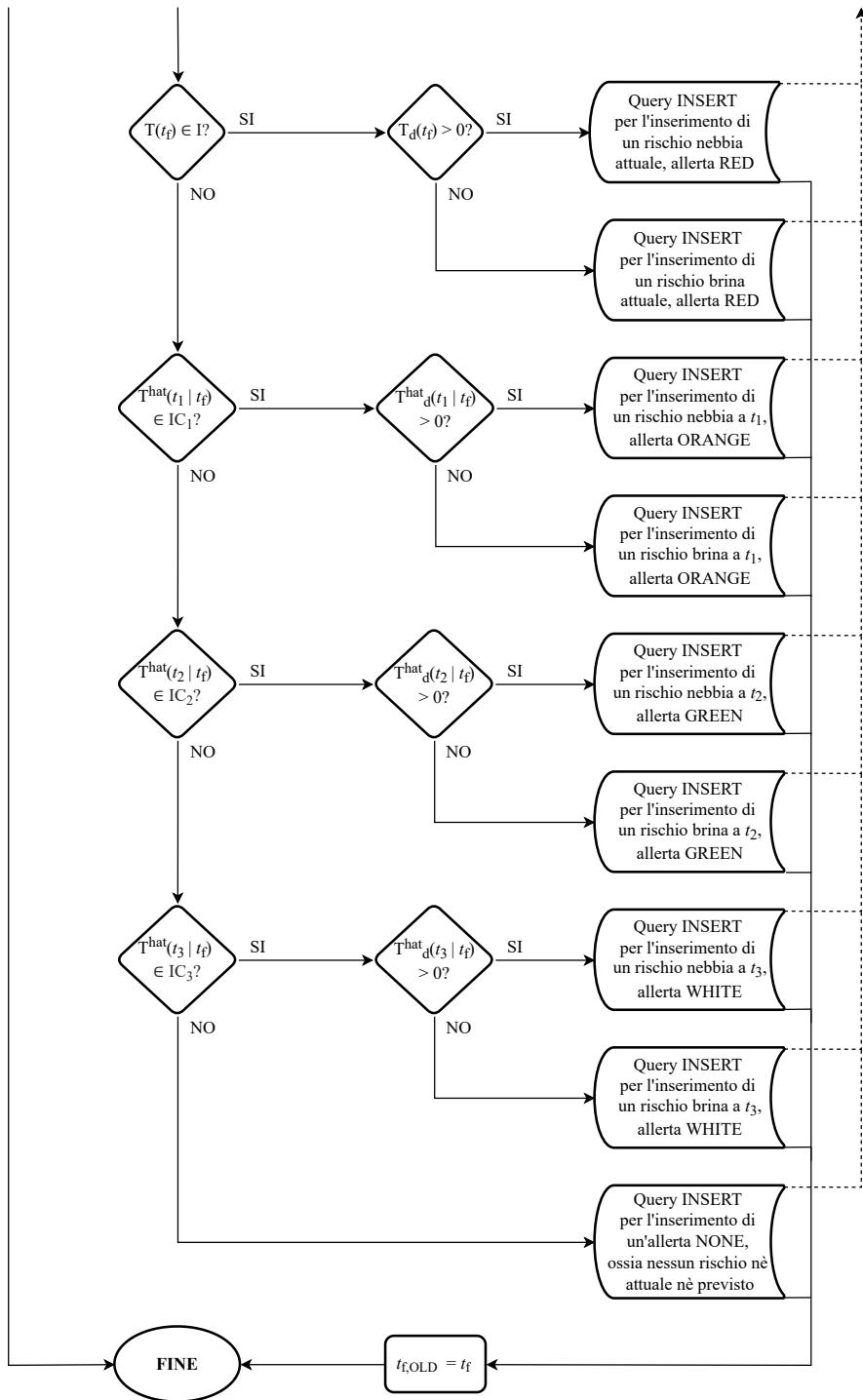


Figura 4.6: Diagramma di flusso relativo all'algoritmo di generazione delle allerte nebbia e brina, seconda parte.

Algoritmo Di seguito sono riportati gli pseudocodici dell'algoritmo e delle funzioni alla base della generazione degli allarmi implementato nell'iterazione 3, descritto tramite i diagramma di flusso precedentemente presentati.

Algoritmo 1: Data Analyzer

```
algoritmo generateAlarms(area_DS, num_areas) is
    /* Avvio dei generatori di allerte */;
    for i = 1 to num_areas do
        badWeatherAlertsCreator(area_DS [i ; 'nameAprStation'], area_DS[i ; 'idArea'], i,
                                old_final_time_list, old_delta_list);
        fogFrostAlertsCreator(area_DS [i ; 'nameAprStation'], area_DS[i ; 'idArea'], i,
                                old_final_time_list);
    end
end
```

Algoritmo 2: generatore di allerte maltempo

```
function badWeatherAlertsCreator(station_code, areaID, index, old_final_time_list,
old_delta_list) is
    recent_time ← data ultima acquisizione della stazione station_code;
    /* Interruzione della funzione se non ci fosse una nuova acquisizione */;
    if recent_time = old_final_time_list [index] then
        | exit;
    end
    /* Elaborazione della serie storica relativa alla pressione atmosferica */;
    pressure_DS ← recupera i dati di pressione atmosferica delle ultime 24 ore della stazione
    station_code;
    if parameter_DS is empty then
        | exit;
    end
    p_mean ← mean(pressure_DS[all ; 'pressure']);
    for i = 1 to lenght(pressure_DS) do
        | pressure_DS[i] ← pressure_DS[i] - p_mean ;           /* Destagionalizzazione */
    end
    pressure_DS ← seleziona pressione ultime 3 ore;
    p_low ← mean(pressure_DS[1:2]);
    p_up ← mean(pressure_DS[end-1:end]);
    delta ← p_up - p_low;
    /* Generazione delle allerte relative al maltempo */;
    if delta ≥ 0 then
        | crea un'allerta NONE, pressione in aumento;
        | inserisci l'allerta nel DB;
    else if delta > old_delta_list [index] then
        | crea un'allerta NONE, maltempo in allontanamento;
        | inserisci tramite una query l'allerta nel DB;
    else if delta < -5 hPa then
        | crea un'allerta RED, maltempo in avvicinamento\picco negativo superato;
        | inserisci l'allerta nel DB;
    else
        | crea un'allerta NONE, pressione in diminuzione;
        | inserisci l'allerta nel DB;
    end
    /* Aggiornamento delle strutture dati di supporto */;
    old_final_time_list [index] ← recent_time;
    old_delta_list [index] ← delta;
end
```

Algoritmo 3: generatore di allerte nebbia e brina, parte 1

```
function fogFrostAlertsCreator(station_code, areaID, index, old_final_time_list) is
    recent_time ← data ultima acquisizione della stazione station_code;
    /* Interruzione della funzione se non ci fosse una nuova acquisizione */;
    if recent_time = old_final_time_list [index] then
        | exit;
    end
    /* Predizione dei valori di temperatura e umidità relativa */;
    summaryT ← crea una tabella per la temperatura con colonne {‘L_low’, ‘value’, ‘L_up’} e
    righe {‘tf’, ‘t1’, ‘t2’, ‘t3’};
    summaryUR ← crea una tabella per l’umidità relativa con colonne {‘L_low’, ‘value’,
    ‘L_up’} e righe {‘tf’, ‘t1’, ‘t2’, ‘t3’};
    summaryTd ← crea una tabella per la temperatura di rugiada con colonne {‘L_low’,
    ‘value’, ‘L_up’} e righe {‘tf’, ‘t1’, ‘t2’, ‘t3’};
    tempURForecaster(‘temperature’,station_code, summaryT);
    tempURForecaster(‘umidity’, station_code, summaryUR);
    /* Calcolo dei valori attuali e previsti della temperatura di rugiada */;
    for i to summaryTd.rows do
        | summaryTd[i ; ‘value’] ← calcolo temperatura di rugiada usando summaryT[i,‘value’]
        |   e summaryUR[i,‘value’];
        | summaryTd[i ; ‘L_up’] ← summaryTd[i ; ‘value’] + 0.45;
        | summaryTd[i ; ‘L_low’] ← summaryTd[i ; ‘value’];
    end
    /* Generazione delle allerte nebbia e brina */;
    alertsCreator(summaryT, summaryTd);
    old_final_time_list [index] ← recent_time;
end
```

Algoritmo 4: generatore di allerte nebbia e brina, parte 2

```
function tempURForecaster(parameter, station_code, dataToUpdate) is
    if parameter = 'temperature' then
        | parameter_DS ← recupera dati di temperatura delle ultime 24 della stazione
        |     station_code;
    else
        | parameter_DS ← recupera dati di umidità delle ultime 24 della stazione station_code;
    end
    if parameter_DS is empty then
        | exit;
    end
    /* Analisi dei dati */;
    model ← fitARIMA(parameter_DS[all ; parameter], start_complexity = [1, 1, 1],
                      stop_complexity = [3, 3, 3]) ;
                      /* Stima del modello ARIMA */
    k ← 3 ;
                      /* Passo della predizione */
    [fc, confint] ← predict(model, k) ;
                      /* Predizione a k passi */
    /* Aggiornamento del dataset */;
    dataToUpdate['tf' ; all] ← parameter_DS[end ; parameter];
    for i = 1 to k do
        | dataToUpdate[i,'value'] ← fc[i];
        | dataToUpdate[i,'I_low'] ← confint[i,'ic_low'];
        | dataToUpdate[i,'I_up'] ← confint[i, 'ic_up'];
    end
end
```

Algoritmo 5: generatore di allerte nebbia e brina, parte 3

```
function alertsCreator(summaryT, summaryTd) is
    if summaryTd ['tf' ; 'I_low'] ≤ summaryT['tf' ; 'value'] ≤ summaryTd ['tf' ; 'I_up']
        then
            if summaryT['tf' ; 'value'] > 0 then
                crea un rischio nebbia attuale, allerta RED;
                inserisci l'allerta nel DB;
            else
                crea un rischio brina attuale, allerta RED;
                inserisci l'allerta nel DB;
            end
        else if summaryTd['t1' ; 'I_low'] ≤ summaryT['t1' ; 'value'] ≤ summaryTd['t1' ; 'I_up']
            then
                if summaryT['t1' ; 'value'] > 0 then
                    crea un rischio nebbia tra 10 min, allerta ORANGE;
                    inserisci tramite una query l'allerta nel DB;
                else
                    crea un rischio brina tra 10 min, allerta ORANGE;
                    inserisci tramite una query l'allerta nel DB;
                end
            else if summaryTd['t2' ; 'I_low'] ≤ summaryT['t2' ; 'value'] ≤ summaryTd['t2' ; 'I_up']
                then
                    if summaryT['t2' ; 'value'] > 0 then
                        crea un rischio nebbia tra 20 min, allerta GREEN;
                        inserisci tramite una query l'allerta nel DB;
                    else
                        crea un rischio brina tra 20 min, allerta GREEN;
                        inserisci tramite una query l'allerta nel DB;
                    end
                else if summaryTd['t3' ; 'I_low'] ≤ summaryT['t3' ; 'value'] ≤ summaryTd['t3' ; 'I_up']
                    then
                        if summaryT['t3' ; 'value'] > 0 then
                            crea un rischio nebbia tra 30 min, allerta WHITE;
                            inserisci tramite una query l'allerta nel DB;
                        else
                            crea un rischio brina tra 30 min, allerta WHITE;
                            inserisci tramite una query l'allerta nel DB;
                        end
                    else
                        crea un'allerta NONE, nessun rischio nè attuale nè previsto;
                        inserisci tramite una query l'allerta nel DB;
                    end
                end
            end
        end
```

Analisi computazionale Procediamo ora con l’analisi della complessità temporale dell’algoritmo.

Nello pseudocodice Alg.1 è riportato l’entry point dell’algoritmo. Infatti, per ogni area vengono chiamate le funzioni *badWeatherAlertsCreator* e *fogFrostAlertsCreator* che si occupano della generazione delle allerte. Queste due funzioni avranno un costo costante per ogni area analizzata. Analizziamo più in dettaglio il costo computazionale delle funzioni:

- **badWheatherAlertsCreator** (Alg.2): questa funzione genera un allarme se è in arrivo una perturbazione, analizzando la pressione atmosferica rilevata dalla stazione associata ad un area. Inizialmente, vengono effettuati dei controlli sui dati disponibili, analizzando con dei semplici costrutti *if*, di costo costante in quanto basati su operazioni elementari di confronto, se i dati disponibili sono già stati analizzati oppure se non sono presenti dei dati nelle ultime 24 ore. Assumiamo che tipicamente questi controlli non determinino la terminazione della funzione, ossia che nel caso peggiore la computazione continui. Successivamente, è presente un ciclo *for* tramite cui viene effettuata la destagionalizzazione. Il ciclo *for* va da 1 a $\text{length}(\text{pressure_DS})$. Tale parametro rappresenta il numero di valori di pressione delle ultime 24 ore di una particolare stazione e, poichè un nuovo valore di pressione viene inserito nel database ogni 10 minuti, tale parametro è mediamente costante e di conseguenza anche la complessità del ciclo risulta essere costante per ogni area. I restanti passaggi sono sequenze di operazioni elementari e costrutti *if-then-else* che hanno una complessità temporale costante, poichè mutualmente esclusivi e composti da operazioni che possono essere considerate costanti in qualunque alternativa scelta.
- **fogFrostAlertsCreator** (Alg.3): questa funzione si occupa della generazione di allerte nebbia e brina, analizzando la temperatura e l’umidità relativa acquisita. Come nella funzione precedentemente descritta, se supponiamo di avere dati sufficienti a disposizione, la funzione esegue operazioni elementari di costo costante e, dal momento in cui viene chiamata una volta per ogni area, non contribuisce ad un aumento della complessità generale. In particolare si noti che la funzione utilizza i dati elaborati da tre sotto funzioni. Vengono effettuate due chiamate della funzione *tempURForecaster* e una chiamata della funziona *alertsCreator*. Le due funzioni sono descritte di seguito. Esse risultano avere complessità costante per ogni area analizzata.
- **tempURForecaster** (Alg.4): è la funzione principale dell’algoritmo, in cui vengono eseguite la stima del modello ARIMA e la predizione a k passi. Inizialmente, vengono recuperati o i dati di temperatura o di umidità della stazione scelta. Successivamente, come nelle funzioni precedenti, viene verificata la disponibilità di dati. Supponendo che tipicamente i dati siano presenti, e quindi riferendoci al caso peggiore, la computazione prosegue stimando il modello ARIMA e la predizione a tre passi della temperatura e umidità. È importante notare che il numero di parametri su cui vengono eseguite queste due operazioni è sempre lo stesso, perchè vengono presi i dati delle ultime 24 ore che si assumono essere costanti per ogni area e la predizione a tre passi può essere considerata costante per ogni modello stimato. Inoltre, la complessità del modello ARIMA stimato è stata limitata. Quindi queste operazioni possono essere considerate costanti per ogni area analizzata. Infine, il ciclo *for* finale eseguirà sempre 3 iterazioni. Di conseguenza, anche il costo di questa funzione è costante.
- **alertsCreator** (Alg.5): questa funzione serve per inserire le allerte nebbia o brina nel database. Non sono presenti cicli, ma solo una sequenza di *if-then-else* mutualmente esclusivi. Per ogni iterazione dell’algoritmo è possibile solamente eseguire una delle operazioni dei rami *if-then-else*, il cui costo computazionale può essere considerata uguale e costante.

Come si è potuto osservare, il costo totale dell’algoritmo può essere considerato $O(n_{\text{areas}})$, dove n_{areas} corrisponde al numero di aree inserite nel database e analizzate. Infatti, si è dimostrato come le due funzioni *badWeatherAlertsCreator* e *fogFrostAlertsCreator* abbiano un costo costante per ogni area analizzata.

4.1.3 Test Data Analyzer

Sono stati eseguiti anche dei test di unità sull'algoritmo. Nella figura 4.7 sono riportati i risultati effettuati grazie al modulo di Python `unittest`. Si è testato la generazione dei seguenti allarmi:

- allarme *RED* per maltempo e allarme *RED* per nebbia;
- allarme *NONE* per maltempo e allarme *RED* per brina;
- allarme *NONE* per maltempo e allarme *NONE* per nebbia;
- allarme *NONE* per maltempo e allarme *WHITE* per nebbia;
- allarme *NONE* per maltempo e allarme *GREEN* per nebbia.

	test (7)	15,7 sec
	test.py (7)	15,7 sec
	TestAlgorithmn (7)	15,7 sec
	test_1_db_initialization	6,5 sec
	test_2_algorithm	8,1 sec
	test_3_fog_RED_and_bw_RED	215 ms
	test_4_frost_RED_and_bw_NONE	215 ms
	test_5_fog_NONE_and_bw_NONE	214 ms
	test_6_fog_WHITE_and_bw_NONE	224 ms
	test_7_fog_GREEN_and_bw_NONE	226 ms

Figura 4.7: Risultati test algoritmo (interfaccia di Visual Studio).

Di seguito è riportato il codice python relativo alla classe che implementa i test. Inizialmente, viene popolato un database di test con i dati caricati da dei file *csv*. All'interno dei file è specificato il nome di una stazione APRS fittizia e i relativi dati di test di temperatura, umidità, pressione e data di acquisizione. Ogni test è associato a una diversa area. Se le operazioni di caricamento termina correttamente, viene lanciata l'esecuzione dell'algoritmo. Se non si verificano eccezioni, vengono poi controllati gli allarmi inseriti dall'algoritmo: se corrispondono agli allarmi attesi, il test è superato.

```

1 class TestAlgorithmn(unittest.TestCase):
2     def test_1_db_initialization(self):
3         try:
4             initialize_database()
5         except:
6             self.fail("Exception in initializing db")
7
8     def test_2_algorithm(self):
9         try:
10             algorithm('testalg', True)
11         except:
12             self.fail("Exception in running algorithm")
13
14     def test_3_fog_RED_and_bw_RED(self):
15         result = get_fog_RED_and_bw_RED()
16
17         expected = [ ('BW', 'RED', 1, 'Maltempo in avvicinamento, delta = -7.0'),
18         ('FOG', 'RED', 1, 'Rischio nebbia attuale')]
```

```

19         self.assertEqual(result, expected)
20
21     def test_4_frost_RED_and_bw_NONE(self):
22         result = get_frost_RED_and_bw_NONE()
23
24         expected = [('BW', 'NONE', 2, 'Pressione in diminuzione, delta = -2.0'),
25 ('FROST', 'RED', 2, 'Rischio brina attuale')]
26
27         self.assertEqual(result, expected)
28
29     def test_5_fog_NONE_and_bw_NONE(self):
30         result = get_fog_NONE_and_bw_NONE()
31
32         expected = [('BW', 'NONE', 3, 'Pressione in diminuzione, delta = -2.0'),
33 ('FOG', 'NONE', 3, 'Nessun rischio n    attuale n    previsto')]
34
35         self.assertEqual(result, expected)
36
37     def test_6_fog_WHITE_and_bw_NONE(self):
38         result = get_fog_WHITE_and_bw_NONE()
39
40         expected = [('BW', 'NONE', 4, 'Pressione in diminuzione, delta = -0.75'),
41 ('FOG', 'WHITE', 4, 'Rischio nebbia tra 30 minuti')]
42
43         self.assertEqual(result, expected)
44
45     def test_7_fog_GREEN_and_bw_NONE(self):
46         result = get_fog_GREEN_and_bw_NONE()
47
48         expected = [('BW', 'NONE', 5, 'Pressione in diminuzione, delta = -0.75'),
49 ('FOG', 'GREEN', 5, 'Rischio nebbia tra 20 minuti')]
50
51         self.assertEqual(result, expected)
52

```

Inoltre, si riporta un esempio di allerta nebbia di tipo *RED* generata dall'algoritmo il 6 febbraio 2022 (Fig.4.8) sulla stazione meteorologica situata nei pressi della località di Ospitaletto (BS). Si è potuto verificare che questa allerta rispecchiasse veramente la situazione meteo attuale. Infatti, le previsioni meteorologiche attuali fornite da Google indicavano la presenza di nebbia (Fig.4.9).



Figura 4.8: Allarme nebbia attuale a Ospitaletto (BS) domenica 06/02/22 alle ore 9:20.

Ospitaletto BS
dom 09:00, Nebbia



Figura 4.9: Meteo attuale fornito da Google a Ospitaletto (BS) domenica 06/02/22 alle ore 9:20.

4.1.4 UC18.1 - Visualizzazione dati ambientali

Breve descrizione: il volontario (o caposquadra) accede alla pagina in cui gli vengono mostrati gli ultimi dati disponibili relativi all'area del team di cui fa parte.

Attori coinvolti: volontario, caposquadra, sistema.

Precondizione: il volontario (o caposquadra) è registrato nel sistema.

Postcondizione: il volontario (o caposquadra) visualizza i dati relativi all'area del team di cui fa parte.

Procedimento:

1. il volontario (o caposquadra) accede alla pagina "Dati Area".
2. il sistema mostra gli ultimi dati disponibili relativi a:
 - stazioni APRS;

4.1.5 UC18.2 - Visualizzazione allarmi

Breve descrizione: il volontario (o caposquadra) accede alla pagina in cui gli vengono mostrati gli allarmi disponibili relativi all'area del team di cui fa parte.

Attori coinvolti: volontario, caposquadra, sistema.

Precondizione: il volontario (o caposquadra) è registrato nel sistema.

Postcondizione: il volontario (o caposquadra) visualizza gli allarmi relativi all'area del team di cui fa parte.

Procedimento:

1. il volontario (o caposquadra) accede alla pagina "Dati Area".
2. il sistema mostra gli allarmi emessi per la specifica area:
 - allarme nebbia o brina;
 - allarme maltempo;

4.1.6 UC10 - Visualizzazione posizione real-time

Breve descrizione: il volontario o il caposquadra accedono alla pagina "Mappa" dove possono vedere la posizione real-time dei soli membri operativi della loro squadra su una mappa.

Attori coinvolti: volontario, caposquadra, sistema.

Precondizione: il volontario/caposquadra è loggato nel sistema.

Postcondizione: il volontario/caposquadra visualizza la posizione dei membri operativi della squadra su una mappa.

Procedimento:

1. il volontario/caposquadra accede alla pagina "Mappa".
2. il sistema recupera le informazioni sulla posizione degli utenti operativi della squadra in modo periodico.
3. il volontario/caposquadra visualizza la posizione dei compagni operativi sulla mappa.

4.2 Component Diagram

Nella figura 4.10 è riportato il diagramma a componenti dell'applicazione dove sono state evidenziate le interfacce e i componenti inseriti nell'iterazione 3.

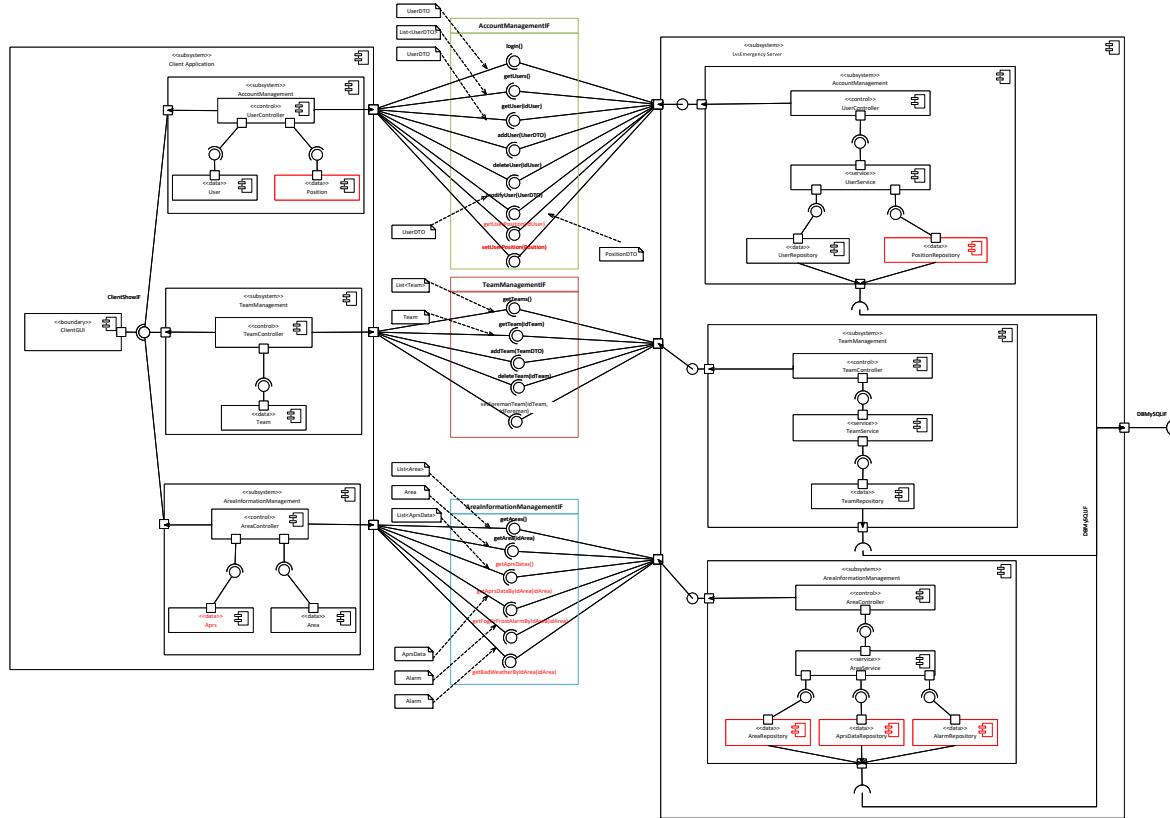


Figura 4.10: Component Diagram.

4.3 Class Diagram

Di seguito è riportata la specifica della struttura dati **PositionDTO**(Fig.4.11). Questo struttura dati è particolarmente utile per incapsulare in un unico oggetto sia le informazioni della posizione che il nome ed il cognome dell'utente a cui appartiene la posizione.

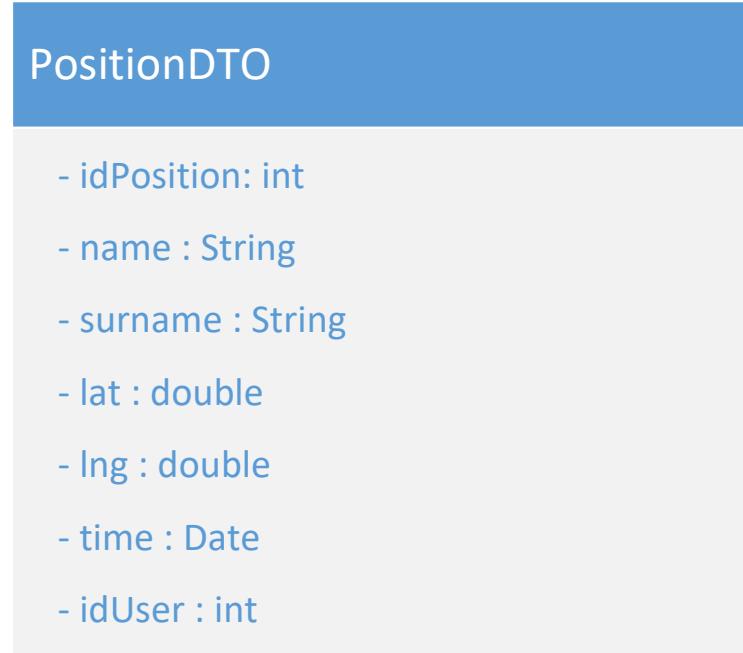


Figura 4.11: Class Diagram **PositionDTO**.

4.4 Interface and Package Diagram

Le API per ricevere i dati meteorologici e gli allarmi sono state implementate nel package *AreaInformationManagement* (Fig.4.12) e sono gestite dal controller **AreaController**. I package *alarm* e *data* sono stati riportati per completezza: al loro interno sono state inserite le classi che gestiscono rispettivamente gli allarmi e i dati provenienti dalle stazioni APRS. Invece, le API che si occupano dell'ottenimento dell'ultima posizione disponibile di un utente attivo e dell'invio della nuova posizione di un utente sono state implementate nel package *AccountManagement*

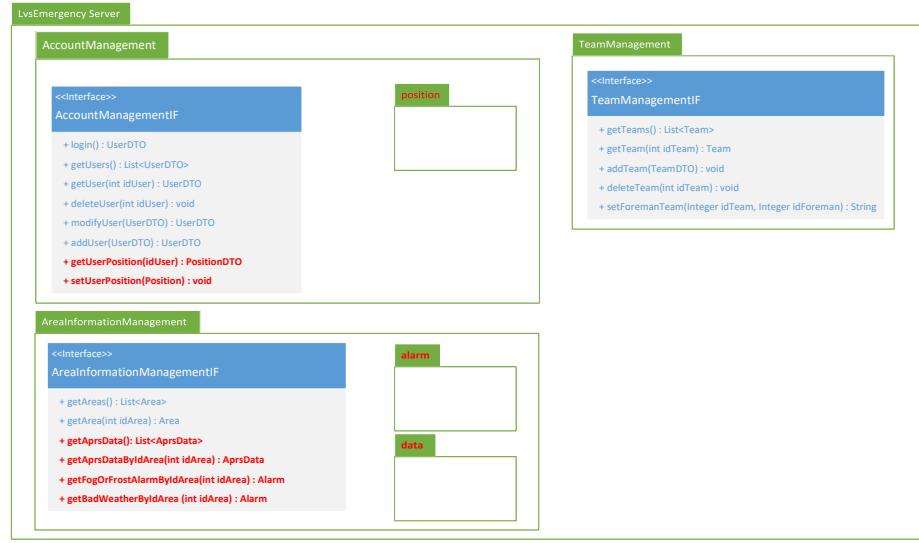


Figura 4.12: Interface and Package Diagram.

4.5 Testing

4.5.1 Analisi statica

Per l'analisi statica del codice Java è stato utilizzato il tool STAN4J, integrato nel IDE Eclipse. Il report è riportato a fine capitolo.

4.5.2 Analisi dinamica

Nell'iterazione 3 si sono testate tutte le API Rest implementate, utilizzando Postman (Fig.4.13). In particolare si sono testate le seguenti funzionalità:

- AreaController:
 - Visualizzazione degli ultimi dati meteo scaricati di un'area, il cui id è specificato nel path della richiesta;
 - Visualizzazione del più recente allarme relativo a nebbia o brina associato all'area il cui id è specificato nel path della richiesta;
 - Visualizzazione del più recente allarme relativo a maltempo associato all'area il cui id è specificato nel path della richiesta;

The screenshot shows three successful API requests from Postman:

- GET Get last aprs data of an area** (localhost:8080/areas/1/aprsdata) - Response: 200 OK, 272 ms, 564 B. Test results: Pass (Codice di stato: 200; Dati aprs ricevuti), Pass (Dati ricevuti corretti).
- GET Get last fog or frost alarm** (localhost:8080/areas/1/alarms/fogorfrost) - Response: 200 OK, 250 ms, 461 B. Test results: Pass (Codice di stato: 200; Allarme ricevuto), Pass (Allarme ricevuto corretto).
- GET Get last bad weather alarm** (localhost:8080/areas/1/alarms/badweather) - Response: 200 OK, 274 ms, 480 B. Test results: Pass (Codice di stato: 200; Allarme ricevuto), Pass (Allarme ricevuto corretto).

Figura 4.13: Risultati test API su Postman.

- UserController:
 - Visualizzazione dell'ultima posizione disponibile di un utente attivo;
 - Visualizzazione dell'ultima posizione disponibile di un utente inattivo;
 - Inserimento della posizione di un utente;

The screenshot shows three API requests in Postman:

- GET Get user position** (localhost:8080/users/4/position) - Status: 200 OK, 202 ms, 491 B. Response: Pass Codice di stato: 200; Posizione utente ricevuta. Pass Dati ricevuti corretti.
- GET Get user position not active** (localhost:8080/users/5/position) - Status: 400 Bad Request, 162 ms, 465 B. Response: Pass Codice di stato: 400; L'utente selezionato non è attivo!
- POST Set user position** (localhost:8080/users/5/position) - Status: 202 Accepted, 297 ms, 375 B. Response: Pass Codice di stato: 202; Posizione inserita correttamente.

Figura 4.14: Risultati test API su Postman.

4.5.3 Unit Test

LvsEmergency Server

In questa iterazione è stata testata la funzione `findFirstByAprsDataIdNameOrderByAprsDataIdTimeDesc(String name)` che recupera i dati più recenti relativi alla stazione APRS, il cui `name` è specificato nel parametro.

Di seguito è riportato il codice del test.

```
1 package it.lvsemergency.areaInformationManagment.data;
2
3 import static org.assertj.core.api.Assertions.assertThat;
4
5 import java.sql.Timestamp;
6
7 import org.junit.jupiter.api.Test;
8 import org.springframework.beans.factory.annotation.Autowired;
9 import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
10 import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
11
12 import it.lvsemergency.areaInformationManagement.data.AprsData;
13 import it.lvsemergency.areaInformationManagement.data.AprsDataId;
14 import it.lvsemergency.areaInformationManagement.data.AprsDataRepository;
15
16 @DataJpaTest
17 @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
18 public class AprsDataRepositoryTest {
19
20     @Autowired
21     private AprsDataRepository underTest;
22
23     @Test
24     void findFirstByAprsDataIdNameOrderByAprsDataIdTimeDesc() {
25
26         Timestamp now = new Timestamp(System.currentTimeMillis());
27         now.setNanos(0);
28
29         AprsDataId aprsDataId = new AprsDataId("IU2CMQ-13", now);
30
31         //given
32         AprsData expected = new AprsData(aprsDataId, 2.8f, 1025.7f, 81, 135f, 0f, 0f, 0f,
33         1.8f, 2.9f, null);
34
35         underTest.save(expected);
36         underTest.findAll();
37
38         //when
39         AprsData result = underTest.findFirstByAprsDataIdNameOrderByAprsDataIdTimeDesc("IU2CMQ-13");
40
41         //then
42         assertThat(expected).isEqualTo(result);
43     }
44 }
```

Il risultato del test con JUnit ha confermato il corretto funzionamento della funzione.

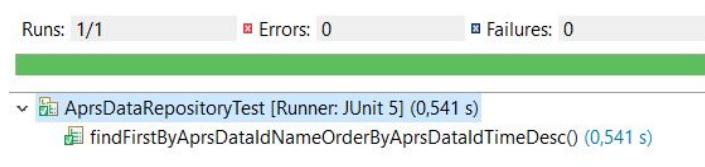


Figura 4.15: Risultato test con JUnit.

Client App

Lato client è stata testata la corretta creazione delle classi aggiunte nell'iterazione 3, `AprsData` e `Alarm` a partire da una stringa JSON. Di seguito è riportato il codice del test:

```
1 void TestEntityIF::testAprsDataFromJson()
2 {
3     QString jsonString = "{\"name\":\"cw6134\", \"time\":\"28-01-2022 01:10:13\", \""
4         "temperature\":0.0, "
5             "\n        \"pressure\":1024.2, \"humidity\":95, \"windDirection\"
6             \":180.0, \"windSpeed\":0.4, "
7                 "\n                \"windGust\":1.3, \"rainOneHour\":0.0, \"rainDay\":0.0, \""
8                     \"rainMidNight\":0.0, \"luminosity\":null}";
9
10    areaInformationManagementIF::AprsData *aprs = new areaInformationManagementIF::
11        AprsData();
12    aprs->fromJsonObject(QJsonDocument::fromJson(jsonString.toUtf8()).object());
13
14    QVERIFY(aprs->getName() == "cw6134");
15    QVERIFY(aprs->getDate() == "28-01-2022 01:10:13");
16    QVERIFY(qFuzzyCompare(aprs->getPressure(), 1024.2f));
17    QVERIFY(qFuzzyCompare(aprs->getTemperature(), 0.0f));
18    QVERIFY(qFuzzyCompare(aprs->getWindDirection(), 180.0f));
19    QVERIFY(qFuzzyCompare(aprs->getWindSpeed(), 0.4f));
20    QVERIFY(qFuzzyCompare(aprs->getWindGust(), 1.3f));
21    QVERIFY(aprs->getHumidity() == 95);
22    QVERIFY(aprs->getRainOneHour() == 0.0);
23    QVERIFY(aprs->getRainDay() == 0.0);
24    QVERIFY(aprs->getRainMidNight() == 0.0);
25
26 void TestEntityIF::testAlarmFromJson()
27 {
28     QString jsonString = "{\"idAlarm\":4, \"time\":\"26-01-2022 18:12:24\", \"type\"
29         "\n        \"FROST\", \"color\":\"GREEN\", \"idArea\":1, \"description\"
30             \":\"no description\"}";
31
32    areaInformationManagementIF::Alarm *alarm = new areaInformationManagementIF::Alarm();
33    alarm->fromJsonObject(QJsonDocument::fromJson(jsonString.toUtf8()).object());
34
35    QVERIFY(alarm->getIdArea() == 1);
36    QVERIFY(alarm->getDate() == "26-01-2022 18:12:24");
37    QVERIFY(alarm->getColor() == "GREEN");
38    QVERIFY(alarm->getType() == "FROST");
39    QVERIFY(alarm->getDescription() == "no description");
40 }
```

Il risultato dei test ha confermato il corretto funzionamento delle nuove funzioni e di quelle precedenti.

```
PacClientAppTest x
01:49:48: Starting C:\Users\David\Desktop\Progetto_PAC\ClientApp\build-PacClientApp-Desktop_Qt_5_15_2_MinGW_64_bit-Debug\test\debug\PacClientAppTest.exe...
***** Start testing of TestEntityIF *****
Config: Using QTest library 5.15.2, Qt 5.15.2 (x86_64-little_endian-llp64 shared (dynamic) release build; by GCC 8.1.0), windows 10
PASS : TestEntityIF::initTestCase()
PASS : TestEntityIF::testUserFromJson()
PASS : TestEntityIF::testTeamFromJson()
PASS : TestEntityIF::testAreaFromJson()
PASS : TestEntityIF::testAprsDataFromJson()
PASS : TestEntityIF::testAlarmFromJson()
PASS : TestEntityIF::cleanupTestCase()
Totals: 7 passed, 0 failed, 0 skipped, 0 blacklisted, 2ms
***** Finished testing of TestEntityIF *****
01:49:48: C:\Users\David\Desktop\Progetto_PAC\ClientApp\build-PacClientApp-Desktop_Qt_5_15_2_MinGW_64_bit-Debug\test\debug\PacClientAppTest.exe exited with code 0
```

Figura 4.16: Risultato test con Qt Test.

Inoltre, è stata testata la corretta creazione della classe Position a partire da una stringa JSON. Di seguito è riportato il codice del test:

```
1 void TestEntityIF::testPositionFromJson()
2 {
3     QString jsonString = "{\"idPosition\": 117, \"lat\": 45.67466793823242, \"lng\": 9.802000190734862, \"time\": \"20-02-2022 14:19:09\", \"idUser\": 16, \"name\": \"Matteo\", \"surname\": \"Verzeroli\"}";
4
5     accountManagement::Position *position = new accountManagement::Position();
6     position->fromJsonObject(QJsonDocument::fromJson(jsonString.toUtf8()).object());
7
8     QVERIFY(position->getIdPosition() == 117);
9     QVERIFY(position->getLat() == 45.67466793823242);
10    QVERIFY(position->getLng() == 9.802000190734862);
11    QVERIFY(position->getDate() == "20-02-2022 14:19:09");
12    QVERIFY(position->getIdUser() == 16);
13    QVERIFY(position->getName() == "Matteo");
14    QVERIFY(position->getSurname() == "Verzeroli");
15 }
16
17 }
```

Il risultato dei test ha confermato il corretto funzionamento delle nuove funzioni e di quelle precedenti.

```
14:29:17: Starting C:\Users\David\Desktop\Progetto_PAC\ClientApp\build-PacClientApp-Desktop_Qt_5_15_2_MinGW_64_bit-Debug\test\debug\PacClientAppTest.exe...
***** Start testing of TestEntityIF *****
Config: Using QTest library 5.15.2, Qt 5.15.2 (x86_64-little_endian-llp64 shared (dynamic) release build; by GCC 8.1.0), windows 10
PASS : TestEntityIF::initTestCase()
PASS : TestEntityIF::testUserFromJson()
PASS : TestEntityIF::testTeamFromJson()
PASS : TestEntityIF::testAreaFromJson()
PASS : TestEntityIF::testAprsDataFromJson()
PASS : TestEntityIF::testAlarmFromJson()
PASS : TestEntityIF::testPositionFromJson()
PASS : TestEntityIF::cleanupTestCase()
Totals: 8 passed, 0 failed, 0 skipped, 0 blacklisted, 2ms
***** Finished testing of TestEntityIF *****
```

Figura 4.17: Risultato test con Qt Test.

4.6 Documentazione API

In questa sezione viene mostrata la documentazione relativa ad alcune API implementate nell'iterazione 3. È possibile visualizzare una collezione (creata con Postman) di tutte le API realizzate e testate sulla repository GitHub. In particolare, si riportano di seguito le più significative:

- API per la visualizzazione degli ultimi dati meteorologici raccolti dalla stazione APRS associata all'area, il cui id è specificato nel path della richiesta;
- API per la visualizzazione del più recente allarme di nebbia o brina generato nell'area di interesse;
- API per la visualizzazione del più recente allarme di maltempo generato nell'area di interesse;
- API per la visualizzazione dell'ultima posizione disponibile di un utente attivo;
- API per l'inserimento della posizione di un utente;

GET Get last aprs data of an area [Open Request →](#)

localhost:8080/areas/1/aprsdata

```
1 {
2   "name": "cw6134",
3   "time": "27-01-2022 17:20:12",
4   "temperature": 3.9,
5   "pressure": 1024.7,
6   "humidity": 87,
7   "windDirection": 180.0,
8   "windSpeed": 0.0,
9   "windGust": 0.9,
10  "rainOneHour": 0.0,
11  "rainDay": 0.0,
12  "rainMidNight": 0.0,
13  "luminosity": null
14 }
```

Figura 4.18: Documentazione API Get Aprs data.

GET Get last fog or frost alarm [Open Request →](#)

localhost:8080/areas/1/alarms/fogorfrost

```
1 {
2   "idAlarm": 4,
3   "time": "26-01-2022 18:12:24",
4   "type": "FROST",
5   "color": "GREEN",
6   "idArea": 1,
7   "description": "Allarme brina. Gravita: VERDE"
8 }
```

Figura 4.19: Documentazione API Get fog or frost alarm.

GET Get last bad weather alarm [Open Request →](#)

localhost:8080/areas/1/alarms/badweather

```

1 {
2     "idAlarm": 1325,
3     "time": "30-01-2022 18:20:36",
4     "type": "BW",
5     "color": "NONE",
6     "idArea": 1,
7     "description": "Pressione in aumento, delta = 1.05"
8 }
```

Figura 4.20: Documentazione API Get bad weather alarm.

GET Get user position [Open Request →](#)

localhost:8080/users/4/position

```

1 {
2     "idPosition": 122,
3     "lat": 45.67466793823242,
4     "lng": 9.802000190734862,
5     "time": "21-02-2022 17:57:58",
6     "idUser": 4,
7     "name": "Giancarlo",
8     "surname": "Bianchi"
9 }
```

Figura 4.21: Documentazione API Get user position.

POST Set user position [Open Request →](#)

localhost:8080/users/5/position

Body:

```

1 {
2     "lat" : 45.67466793823242,
3     "lng" : 9.802000190734862
4 }
```

Response: Position set correctly!

Figura 4.22: Documentazione API Set user position.

Quality Report

Creation Date 2022-02-20

Package Prefix it.

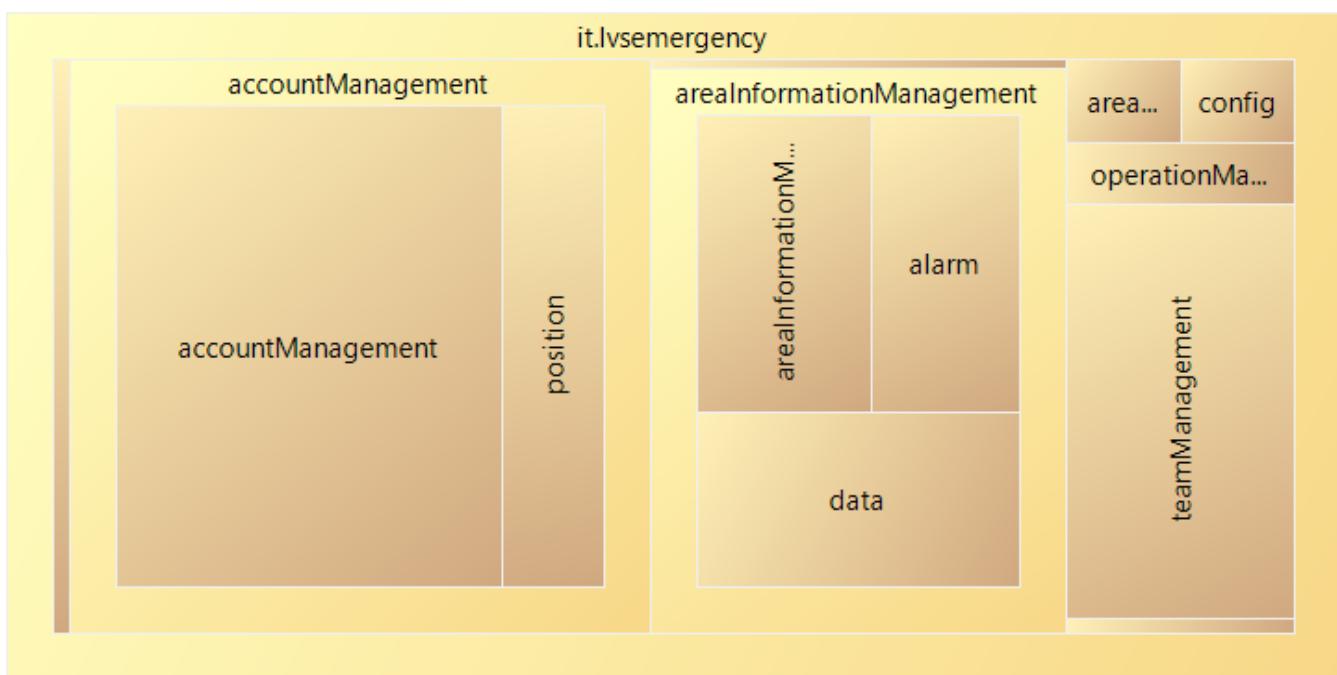
Level of Detail Member

Tmp_220123_212257

Library Dependency Graph



Treemap Overview



Metrics Summary

Metric	Value
Number of Libraries	3
Number of Packages	12
Number of Top Level Classes	37
Average Number of Top Level Classes per Package	3.08
Average Number of Member Classes per Class	0
Average Number of Methods per Class	7.97
Average Number of Fields per Class	2.76
Estimated Lines of Code	1298
Estimated Lines of Code per Top Level Class	35.08

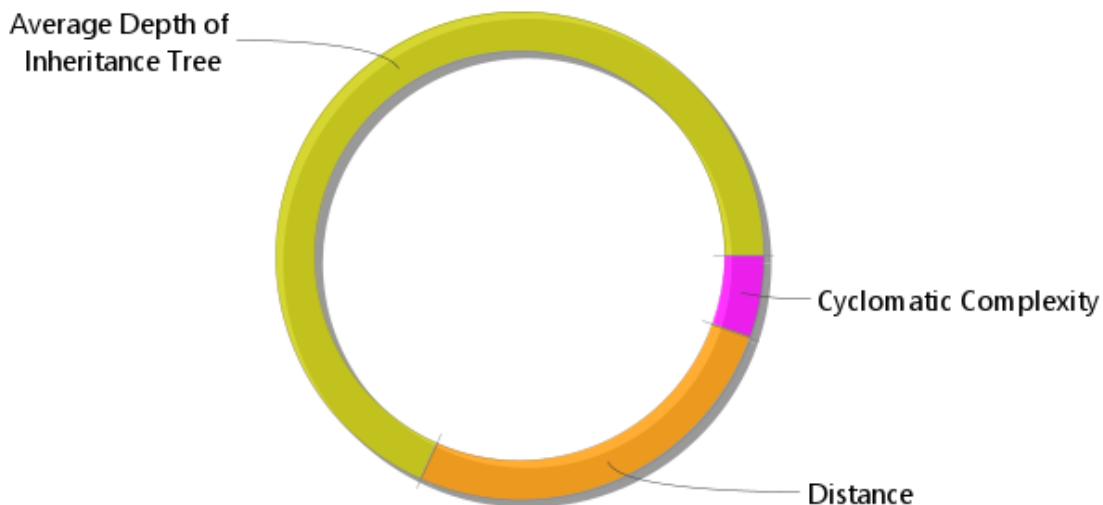
Average Cyclomatic Complexity	1.03
Fat for Library Dependencies	1
Fat for Flat Package Dependencies	7
Fat for Top Level Class Dependencies	76
Tangled for Library Dependencies	0%
Average Component Dependency between Libraries	16.67%
Average Component Dependency between Packages	8.33%
Average Component Dependency between Units	9.91%
Average Distance	0.01
Average Absolute Distance	0.39
Average Weighted Methods per Class	8.19
Average Depth of Inheritance Tree	0.86
Average Number of Children	0
Average Coupling between Objects	1.19
Average Response for a Class	9.76
Average Lack of Cohesion in Methods	28.92

Top Violations (8 of 8)

Artifact	Metric	Value
Tmp_220123_212257	DIT	0.86
lvsemergency.areaInformationManagement.data	D	-0.67
lvsemergency.areaInformationManagement.alarm	D	-0.75
lvsemergency.accountManagement.position	D	-0.67
lvsemergency.accountManagement.User.equals(...)	CC	17
lvsemergency.areaInformationManagement.data.AprsData.equals(...)	CC	15
lvsemergency.operationManagement	D	1
lvsemergency.alarmNotificationManagement	D	1

Pollution Chart

Pollution 0.55



Violations by Metric

Cyclomatic Complexity

Artifact	Value
lvsemergency.accountManagement.User.equals(...)	17
lvsemergency.areaInformationManagement.data.AprsData.equals(...)	15

Distance

Artifact	Value
lvsemergency.areaInformationManagement.data	-0.67
lvsemergency.areaInformationManagement.alarm	-0.75
lvsemergency.accountManagement.position	-0.67
lvsemergency.operationManagement	1
lvsemergency.alarmNotificationManagement	1

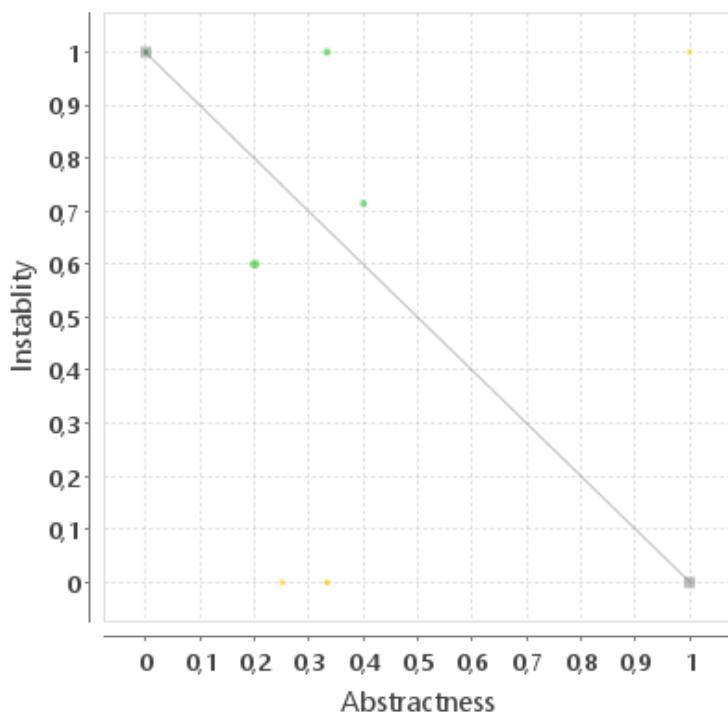
Average Depth of Inheritance Tree

Artifact	Value
Tmp_220123_212257	0.86

Design Tangles

There are no design tangles.

Package Distance Chart



Metric Ratings

Count Metrics

Metric	Rating	Linear
Number of Top Level Classes	20	✓

- Number of Methods
- Number of Fields
- Estimated Lines of Code
- Estimated Lines of Code



Complexity Metrics

Metric	Rating	Linear
Cyclomatic Complexity		✓
Fat		✓
Fat		✓
Fat		✓
Tangled		✓
Tangled for Library Dependencies		✓
Average Component Dependency between Libraries		✓
Average Component Dependency between Packages		✓

Robert C. Martin Metrics

Metric	Rating	Linear
Distance		✓
Average Absolute Distance		✓

Chidamber & Kemerer Metrics

Metric	Rating	Linear
Weighted Methods per Class		✓
Depth of Inheritance Tree		✓
Average Depth of Inheritance Tree		✓
Coupling between Objects		✓
Response for a Class		✓

Capitolo 5

Manuale Utente

Grazie alle scelte tecnologiche con cui sono stati implementati i diversi componenti, il prodotto software può essere utilizzato ovunque e sulla maggior parte dei sistemi operativi, sia desktop (Windows, MacOS, Linux), sia mobile (Android o iOS), rispettando il requisito non funzionale della portabilità richiesto dal cliente. L'applicativo si compone di quattro macro componenti:

- **Web Server Spring:** questa componente è in esecuzione su una istanza di Azure Spring Cloud e di conseguenza è sempre accessibile dall'applicazione client;
- **Database MySQL:** anch'esso in esecuzione su una istanza di Azure MySQL;
- **Data Collector e Data Analyzer:** queste due applicazioni sono in esecuzione su un Raspberry Pi 4;
- **Applicazione Client:** grazie al framework Qt, tale applicazione può essere compilata ed eseguita su gran parte dei sistemi operativi. Infatti, l'unica cosa che deve fare l'utente è installare l'APK sul proprio telefono, oppure installare l'applicazione sul proprio computer e lanciarla all'occorrenza. In questo modo, l'utente può accedere ed utilizzare l'applicazione ed i servizi offerti in qualunque momento.

In figura 5.1a viene mostrata la schermata di login dell'applicazione. Per accedere alle funzionalità, l'utente deve inserire lo username e la password che gli sono stati fornite dal coordinatore. Dopo aver inserito le credenziali, l'utente accede alla dashboard (figura 5.1b) in cui è possibile osservare alcune informazioni sul proprio account ed una mappa che mostra la posizione dell'area di appartenenza. Nel menù laterale è possibile scegliere quali funzionalità utilizzare (figura 5.1c): si possono visualizzare informazioni meteorologiche (figura 5.1e), allarmi (figura 5.1d) o informazioni sul proprio account e sul proprio team, compresi alcuni dati sui membri del team (figura 5.1f).

Il coordinatore potrà invece scegliere dei servizi diversi da quelli di un utente normale (figura 5.2a). Infatti, tale figura ha il compito di inserire nuovi volontari inserendone i dati (figura 5.2b), creare nuove squadre e cancellare degli utenti. Da rileggere

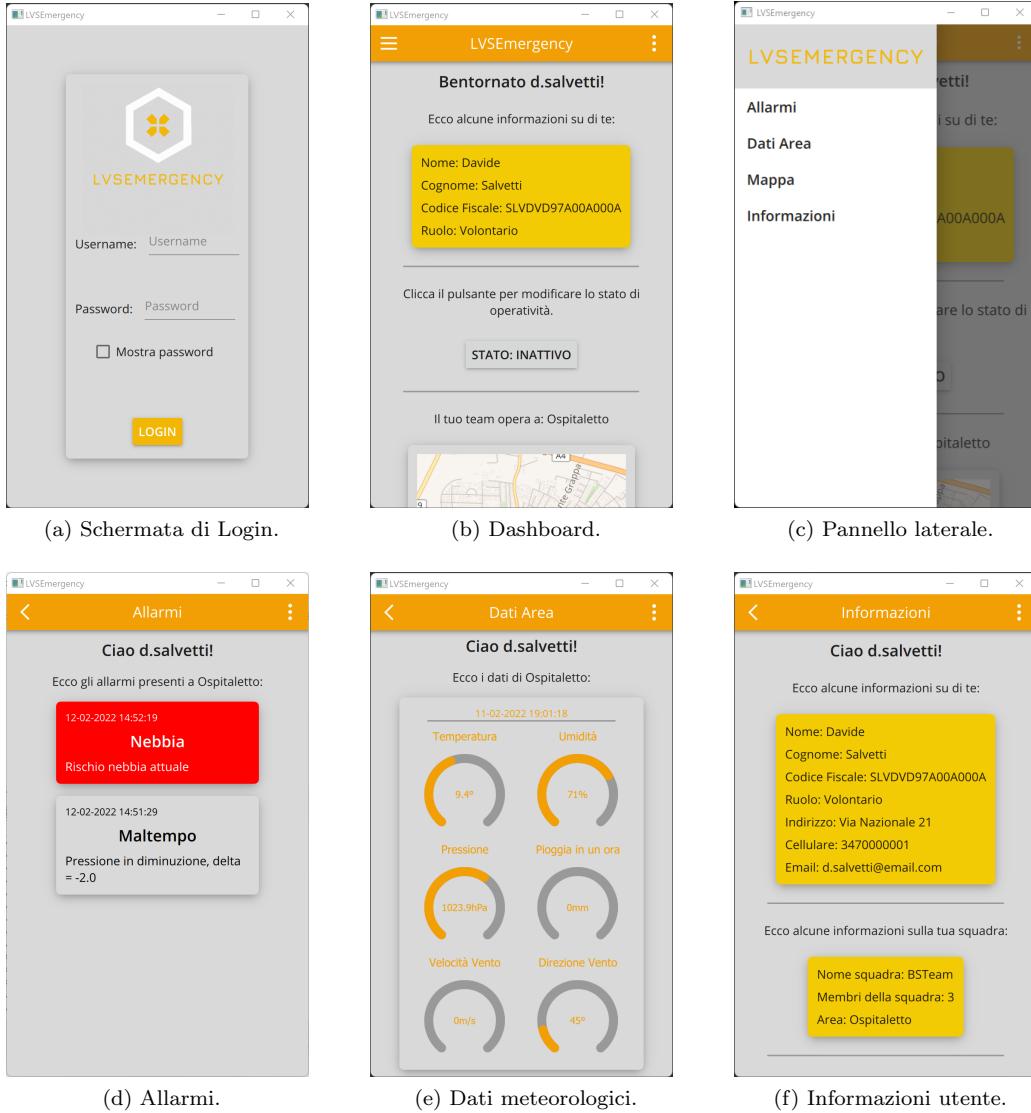
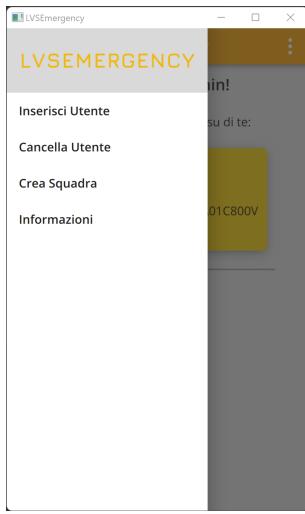


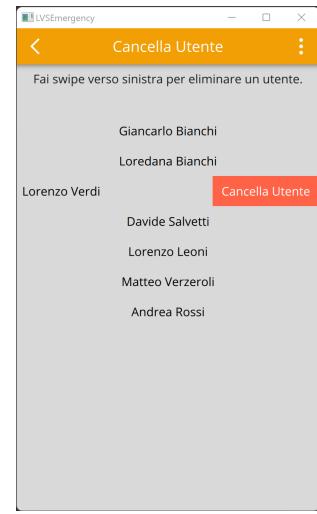
Figura 5.1: Funzionalità accessibili da volontari e caposquadra.



(a) Pannello laterale del coordinatore.

A screenshot of the "Inserisci Utente" (Insert User) screen. It features a back arrow, a title bar with "Inserisci Utente", and a three-dot menu icon. The form contains fields for "Username", "Nome", "Cognome", "Password" (with a "Mostra password" checkbox), "Codice Fiscale", "Sesso" (with a dropdown menu showing "Maschio"), "Indirizzo", "Cellulare", and "Email".

(b) Inserisci utente.



(c) Cancella utente.

Figura 5.2: Funzionalità accessibili dai coordinatori.

Capitolo 6

Conclusioni

6.1 Sviluppi futuri

Non tutti i casi d'uso definiti sono stati implementati. Di seguito è riportata una tabella riassuntiva di quali casi d'uso sono stati implementati e quali no.

Codice	Caso d'uso	Implementato
Alta Priorità		
UC1	Login	Sì
UC2	Logout	Sì
UC3	Visualizzazione informazioni account	Sì
UC4	Visualizzazione informazioni squadra	Sì
UC15	Inserimento utente	Sì
UC16	Cancellazione utente	Sì
UC17	Gestione squadre (creazione)	Sì
UC18	Visualizzazione informazioni zona	Sì
Media Priorità		
UC6	Segnalazione operatività	Sì
UC7	Visualizzazione intervento di emergenza	No
UC8	Visualizzazione intervento programmato	No
UC9	Inserimento informazioni intervento	No
UC11	Gestione intervento di emergenza	No
UC12	Gestione intervento programmato	No
UC13	Gestione report intervento di emergenza	No
UC14	Gestione report intervento programmato	No
UC20	Gestione informazioni relative alla zona	No
Bassa Priorità		
UC5	Gestione reperibilità	No
UC10	Visualizzazione posizione real-time	Sì
UC19	Notifiche allarmi zona	No

Tabella 6.1: Casi d'uso implementati e mancati.

6.1.1 API Protezione Civile POP

Di seguito si riporta un esempio di chiamata dell'API Protezione Civile POP che permette di visualizzare alcuni allarmi generati dalla Protezione Civile italiana (Fig.6.1), presentata nei requisiti ma non implementata in questo progetto. La documentazione è disponibile sul sito <https://www.protezionecivilepop.tk/>.

GET API Protezione Civile POP [Open Request →](#)

`https://www.protezionecivilepop.tk/allerte?citta=016024&rischio=idraulico&allerta=verde&giorno=domani&formato=json`

Make things easier for your teammates with a complete request description.

Query Params

citta	016024
rischio	idraulico
allerta	verde
giorno	domani
formato	json

Response:

```
1 {  
2   "titolo": "Previsioni del Dipartimento Protezione Civile - Bollettini di criticità",  
3   "descrizione": "RSS generato dal servizio online \"Protezione Civile POP\"",  
4   "url": "https://github.com/pdm-dpc/DPC-Bollettini-Criticità-Idrogeologica-  
      Idraulica/blob/master/files/all/latest_all.zip",  
5   "lingua": "it-IT",  
6   "license": "Creative Commons BY-SA 4.0",  
7   "previsione": {  
8     "date": "Tue, 22 Feb 22 00:00:00 +0100",  
9     "risk": "idraulico",  
10    "info": "Assenza di criticità",  
11    "alert": "BIANCA",  
12    "city_name": "Bergamo",  
13    "province_name": "Bergamo",  
14    "province_code": "BG",  
15    "region": "Lombardia",  
16    "latitude": 45.6833,  
17    "longitude": 9.7167,  
18    "istat_code": "016024",  
19    "civil_protection_zone_id": "Lomb-10",  
20    "civil_protection_zone_info": "Lomb-10",  
21    "link": "https://github.com/pdm-dpc/DPC-Bollettini-Criticità-Idrogeologica-  
      Idraulica/blob/master/files/all/latest_all.zip",  
22    "date_publication": "Mon, 21 Feb 22 15:31:00 +0100"  
23  }  
24 }
```

Figura 6.1: Documentazione API Protezione Civile POP.

6.1.2 API Terremoti

Nella figura 6.2 è presentata un esempio di API che permette di visualizzare i terremoti registrati dall'Istituto Nazionale di Geofisica e Vulcanologia, presentata nei requisiti ma non implementata nell'applicazione. La documentazione completa può essere consultata al link <https://developers.italia.it/api/terremoti-opendata>.

GET API Terremoti [Open Request →](#)

```
http://webservices.ingv.it/fdsnws/event/1/query?boundaries-rect&maxlat=60&minlon=13&maxlon=15&format=text&minlat=4
```

Make things easier for your teammates with a complete request description.

Query Params

boundaries-rect	
maxlat	60
minlon	13
maxlon	15
format	text
minlat	4

Response:

```
1 #EventID|Time|Latitude|Longitude|Depth/Km|Author|Catalog|Contributor|ContributorID|
   MagType|Magnitude|MagAuthor|EventLocationName|EventType
2 30020691|2022-02-22T10:29:00.190000|44.7875|10.7798|8.2|SURVEY-INGV||||ML|2.6|--|2 km
   N Correggio (RE)|earthquake
```

Figura 6.2: Documentazione API Terremoti.

6.2 Approfondimento - Qt

Qt è un framework per lo sviluppo di applicazioni cross-platform: con un unico codice è possibile creare applicazioni in grado di girare su diversi sistemi operativi, come Windows, MacOS, Android e iOS. Il linguaggio su cui si basa è il C++ ma è possibile utilizzare anche altri linguaggi, come Python e Java. La potenzialità del framework Qt è la sua architettura modulare con un elevato livello di astrazione rispetto la piattaforma sottostante. Il modulo principale, *QtCore*, include l'implementazione di svariate strutture dati, inclusi tipi generici e classi apposite per la manipolazione delle stringhe di testo. Per la creazione di interfacce grafiche, uno dei componenti principali è il *Qt Widget*, che viene utilizzato principalmente per lo sviluppo in ambito Desktop. Per applicazioni embedded, invece, viene utilizzato il modulo *QtQuick* che garantisce una maggiore separazione tra le entità coinvolte nell'implementazione dell'interfaccia grafica e quelle che riguardano la *business logic* dell'applicazione.

In questo progetto è stato utilizzato il C++ come linguaggio di programmazione per la *business logic*, mentre per la parte grafica è stato utilizzato il modulo *QtQuick* con il *QML (Qt Modelling Language)*, un linguaggio dichiarativo basato su JavaScript.

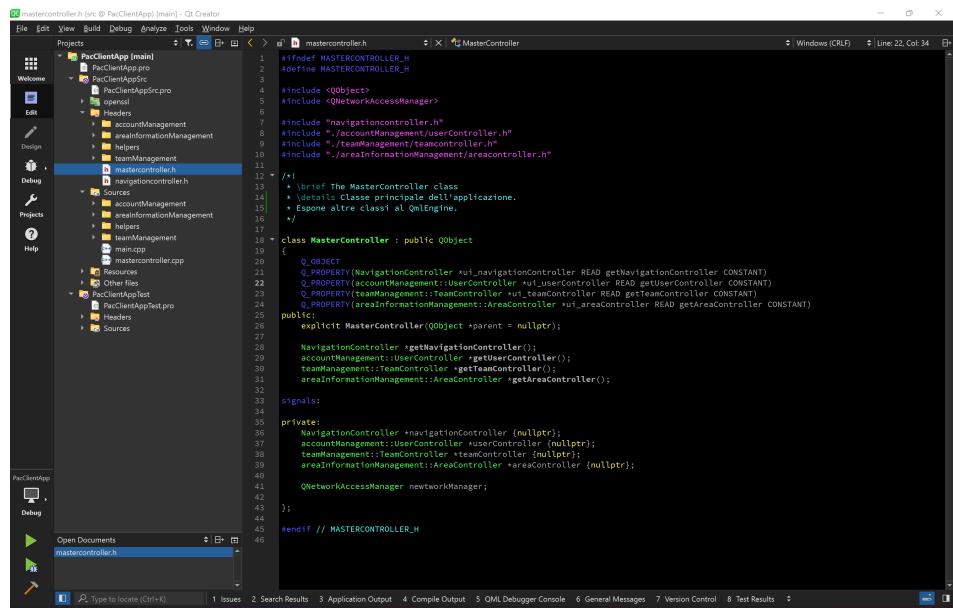


Figura 6.3: Schermata dell'IDE *Qt Creator* con il progetto aperto.

Il progetto è stato strutturato come mostrato in figura 6.4. È stato creato un progetto principale, nominato *PacClientApp*, che contiene due sotto progetti: *PacClientAppSrc* è il progetto che contiene i sorgenti relativi all'applicazione, mentre *PacClientAppTest* è il progetto che contiene i file per eseguire i test automatici grazie al modulo *QtTest*. All'interno di *PacClientAppSrc*, sono state create delle sotto cartelle per la gestione dei *namespace* in C++, che sono l'equivalente dei *package* in Java. Le sottocartelle create sono le stesse inserite lato server, e riflettono l'architettura dell'applicazione. L'unico *namespace* estraneo è *lhelpers*, che contiene alcune classi con delle funzioni di supporto che vengono utilizzate in diversi punti dell'applicazione.

All'interno dei *namespace* è stata creata una componente *controller* che si occupa di effettuare le richieste tramite API REST al server e di gestire le risposte.

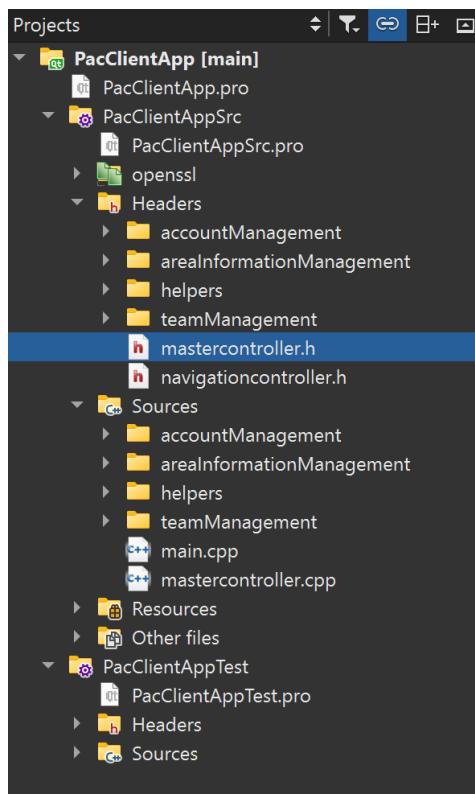


Figura 6.4: Schermata dell'IDE *Qt Creator* con il progetto aperto.