Manuele Tecnico

Samuele Moranzoni 754159 VA , Edoardo Di Tullio 753918 VA

Universita' degli Studi dell'Insubria– Laurea Triennale in Informatica Progetto Laboratorio B: Climate Monitoring Versione: febbraio 2024

Climate Monitoring Application

Laboratorio b

Contents

1 Introduzione

Il progetto Climate Monitoring è stato sviluppato nell'ambito del corso di Laurea in Informatica presso l'Università degli Studi dell'Insubria. Il sistema monitora luoghi geografici fornendo informazioni specifiche tramite un database centralizzato.
Il progetto si basa su due componenti principali:

- File codice: regolano il funzionamento dell'applicazione e le azioni visibili all'utente.
- Database(SERVERLABB): memorizza le credenziali degli operatori registrati , i centri di monitoraggio e le informazioni sui luoghi geografici oggetto di monitoraggio .

N.B: eventuali operazioni come avvio , esecuzione , troubleshooting del progetto sono trattate nel manuale utente.

2 Librerie esterne

- java.desktop
- java.naming
- java.net
- java.rmi
- java.swing
- java.sql
- java.xm Nello specifico ogni libreria esterna contribuisce cosi':

Nello specifico ogni libreria esterna contribuisce cosi':

1. java.desktop

   Include funzionalità per creare applicazioni desktop con interfacce grafiche (GUI). Contiene le API per gestire finestre, pulsanti, campi di testo e altre componenti visive, oltre a supportare eventi legati all'interfaccia utente.

2. java.naming

   Fornisce un'interfaccia per interagire con i servizi di naming, come quelli basati su JNDI (Java Naming and Directory Interface). Viene usata per trovare e accedere a risorse come database, messaggistica o server attraverso nomi e directory

3. java.net

   Gestisce le operazioni di rete, come la creazione di socket, la gestione delle connessioni di rete (HTTP, FTP, ecc.), e la gestione delle risorse web. È alla base della comunicazione tra applicazioni distribuite tramite Internet o reti locali.

4. java.rmi

   RMI (Remote Method Invocation) è una libreria per la creazione di applicazioni distribuite in cui un programma Java può chiamare metodi su oggetti situati su macchine remote. Permette la comunicazione tra oggetti Java in ambienti distribuiti.

5. java.swing

   E' una libreria che fa parte della Java Foundation Classes (JFC), ed è utilizzata per la creazione di interfacce grafiche utente (GUI) in applicazioni desktop Java. Fornisce una serie di componenti e strumenti per costruire finestre, pulsanti, caselle di testo, etichette e altri elementi grafici.

6. java.swing

   E' una libreria che fa parte della Java Foundation Classes (JFC), ed è utilizzata per la creazione di interfacce grafiche utente (GUI) in applicazioni desktop Java. Fornisce una serie di componenti e strumenti per costruire finestre, pulsanti, caselle di testo, etichette e altri elementi grafici.

7. java.sql

   Contiene le API per interagire con database SQL. Supporta operazioni come connessioni ai database, esecuzione di query SQL, aggiornamenti e gestione dei risultati, ed è la libreria principale per la gestione dei dati nelle applicazioni aziendali.

8. java.xml

   Contiene le API per interagire con database SQL. Supporta operazioni come connessioni ai database, esecuzione di query SQL, aggiornamenti e gestione dei risultati, ed è la libreria principale per la gestione dei dati nelle applicazioni aziendali.

3 Funzionalità offerte da ClimateMonitoring

Il prossimo diagramma UML rappresenta quelle che sono le funzionalità che l'applicativo offre , è importante discernere quali funzionalità siano destinate agli operatori e quali ai cittadini comuni. Sono stati rispettati i seguenti requisiti : Tutti possono:

- cercare aree tramite nome, stato o coordinate geografiche
- visualizzare i parametri climatici associati a ciascuna area di interesse

Gli operatori autorizzati possono:

- Registrarsi/loggarsi all'applicazione
- creare centri di monitoraggio con l'elenco delle aree di interesse
  N.B.: Abbiamo deciso di fare in modo che prima un utente crei il suo centro di monitoraggio (solo se quest'ultimo non è già registrato) e dopo possa aggiungere le aree : associandole al proprio da una lista con tutte le aree registrate nel database (si è deciso che un' area puo' esser monitorata da piu' centri) oppure creandole se non esistono.

- inserire i valori dei parametri climatici per un'area di interesse

Per quanto riguarda l' attore ServerAdmin si è rispettato codesto requisito : Al lancio di serverCM deve essere richiesto di specificare:

1. le credenziali per accedere al dbCM (database di supporto all'esecuzione dei servizi della piattaforma CM)
2. l'host del DB

Le funzionalità sono esplicate tramite diagrammi dinamici UML nella sezione 6.

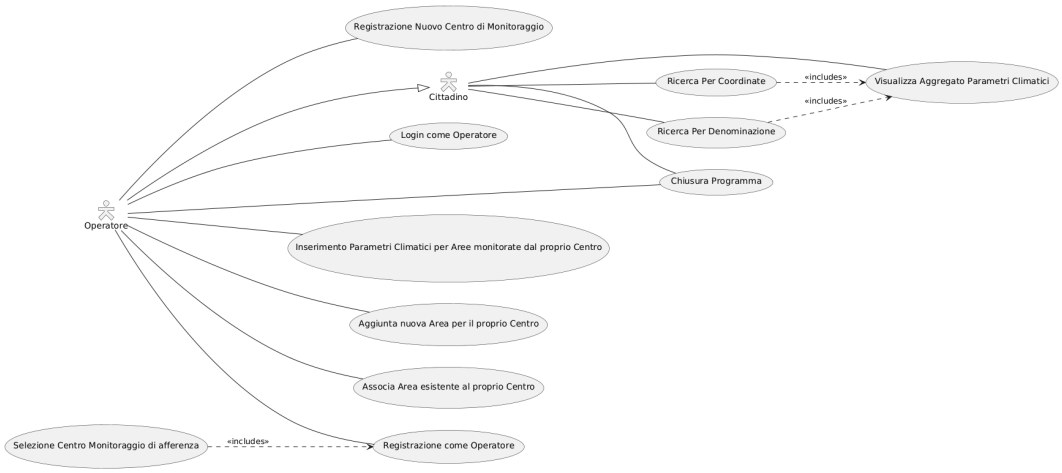

Figure 1: Use Case diagram: Cittadino ed Operatore , Operatore specializza Cittadino

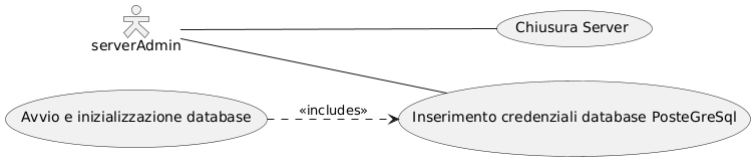

Figure 2: Use Case diagram:Server Admin

4 Logica di comunicazione del progetto

L'applicazione è strutturata in 4 componenti principali: il **clientCM** che riceve richieste da parte dell'utente , il **serverCM** che a suo volta possiede e controlla un' istanza di una classe denominata **Database Connection** che sarà fondamentale per gestire le richieste da parte del client e le risposte da parte del database **SERVERLABB**. Quando attivo, il **ServerCM** inizializza un'istanza della classe **DatabaseConnection** e la passa come input all'oggetto **RemoteServiceImpl**, che implementa i metodi di **DatabaseConnection**. Questo oggetto viene quindi pubblicato nel registro RMI, diventando accessibile come oggetto remoto. Il **ClientCM**, utilizzando l'architettura RMI, può ottenere il riferimento all'oggetto remoto, interrogare in modo sicuro il database **SERVERLABB** e ricevere le

risposte basate sui dati forniti dall'utente. Qua di seguito viene mostrato un estratto del contenuto logico di **ServerRMI**

```
    // L'oggetto usato dal server per connettersi al database si connette
            DatabaseConnection dc = new DatabaseConnection();
            dc.connetti();
            // Crea l'oggetto remoto
            RemoteServiceImpl remoteService = new RemoteServiceImpl(dc);

            // Crea il registro RMI sulla porta 1099
            LocateRegistry.createRegistry(1099);

            // Registra l'oggetto remoto nel registro
            Naming.rebind("rmi://localhost/climatemonitoring.RemoteService",
                remoteService);

            System.out.println("Server RMI avviato e in ascolto..." +
                               "se vuoi interrompere la connessione scrivi 'stop'
                                    nella prossima riga ");
```

4.1 Architettura RMI: Server , DatabaseConnection e RemoteServiceImpl

L'architettura RMI in questo caso permette al client remoto **ClientCM** di accedere ai dati di un database centrale **SERVERLABB** tramite il server RMI. Il server avvia un registro RMI sulla porta 1099, autentica le credenziali del database, stabilisce la connessione di **Database Connection** e pubblica l'oggetto remoto RemoteServiceImpl nei registri preso in input **Database Connection**, che include la logica per interagire con il database . Il client, ottenendo un riferimento a questo oggetto remoto dal registro, può invocare metodi definiti nell'interfaccia remota come se fossero locali, inviando richieste al server per interrogare il database. Questa architettura decoupla client e server, centralizza i dati e garantisce trasparenza nella comunicazione remota. Un esempio di utilizzo dell'oggetto remoto da parte del client per la funzione di registrazione nel database dell'utente puo' essere :

```
    private void performLogin() {
       try {
            String userid = useridField.getText();
            String password = new String(passwordField.getPassword());

            if (userid.isEmpty() || password.isEmpty()) {
                JOptionPane.showMessageDialog(this,
                        "Inserisci tutti i campi",
                        "Campi vuoti",
                        JOptionPane.INFORMATION_MESSAGE);
                return;
            }

```

```
14              // Connessione al servizio remoto
15              Registry registry = LocateRegistry.getRegistry("localhost", 1099);
16              RemoteService stub = (RemoteService) registry.lookup("
                    climatemonitoring.RemoteService");
17              OperatoreRegistrato operatore = stub.loginOperatore(userid,
                    password);
18
19              if (operatore != null && operatore.getId() > 0) {
20                  OperatoreSession.getInstance().setOperatore(operatore);
21                  JOptionPane.showMessageDialog(this,
22                          "Login avvenuto con successo! Benvenuto, " + userid);
23                  new AreaRiservataOperatorFrame();
24                  dispose();
25              } else {
26                  JOptionPane.showMessageDialog(this,
27                          "Credenziali errate: reinseriscile",
28                          "Login fallito",
29                          JOptionPane.INFORMATION_MESSAGE);
30              }
```

N.B: gli oggetti serializzati scambiati tra client e oggetto remoto possono essere inizializzati con valori id emblematici in caso di esito negativo dell'operazione offerta : nel caso di buon esito dell'operazione l'entità ha valore id positivo (che è il valore id seriale chiave primaria) , nel caso di esito negativo l'entità è inizializzata con valore negativo cosicchè il clientCM con i suoi frame possa riconoscere il valore negativo specifico e dunque poter esplicare all'utente la natura dell'errore che è causa dell'esito negativo dell'operazione . Nel codice l'istanza Operatore Registrato è inizializzato con un id negativo se il login non va a buon fine , il client controlla e avvisa l'utente dell'operazione di login fallita.

```
1     /**
2      * Permette il login di un operatore, restituendo l'istanza di
           climatemonitoring.OperatoreRegistrato.
3      *
4      * @param userid   l'userid dell'operatore.
5      * @param password la password dell'operatore.
6      * @return l'oggetto climatemonitoring.OperatoreRegistrato in caso di
           successo,
7      *         un oggetto con id negativo in caso di errore.
8      */
9     public synchronized OperatoreRegistrato loginOperatore(String userid,
          String password) {
10        String sql = "SELECT id, centro_monitoraggio_id FROM
              OperatoriRegistrati WHERE userid = ? AND password = ?";
11
12        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
13
14            pstmt.setString(1, userid);
```

```
15              pstmt.setString(2, password);
16
17              try (ResultSet rs = pstmt.executeQuery()) {
18                  if (rs.next()) {
19                      int id = rs.getInt("id");
20                      Integer centro_monitoraggio_id = (Integer) rs.getObject("
                            centro_monitoraggio_id");
21
22                      System.out.println("Login effettuato con successo per l'
                            operatore: " + userid + " con ID: " + id +
23                                ", Centro Monitoraggio ID: " + (
                                      centro_monitoraggio_id != null ?
                                      centro_monitoraggio_id : "non assegnato ( di
                                      tipo null )"));
24
25                      return new OperatoreRegistrato(id, centro_monitoraggio_id,
                             userid);
26                  } else {
27                      System.err.println("Login fallito: userid o password
                            errati.");
28                      return new OperatoreRegistrato(-1);
29                  }
30              }
31
32          } catch (SQLException e) {
33              System.out.println("Errore nel tentativo di connessione per
                    effettuare il login: " + e.getMessage());
34              e.printStackTrace();
35              return new OperatoreRegistrato(-2);
36          }
37      }
```

5 Struttura statica del progetto

Il codice sorgente è sviluppato in 3 moduli principali : common , clientCM , serverCM . Come da specifica , ognuno dei 3 moduli è sviluppato all'interno del package **climatemonitoring** preceduta dalla struttura tipica di un progetto compilato con Maven : **src/main/java** . Il progetto comprende anche il database PostgreSql SERVERLABB , sviluppato come prima componente del progetto .

5.1 ServerCM

Il modulo `serverCM` è responsabile della gestione del server per il sistema di monitoraggio climatico, consentendo comunicazioni remote al client tramite RMI (Remote Method Invocation). Esso include diverse classi principali:

5.1.1 DatabaseConnection

Gestisce la connessione al database PostgreSQL, fornendo metodi per autenticazione, inserimento e recupero di dati.

5.1.2 RemoteService

Definisce l'interfaccia dei metodi remoti che possono essere invocati dai client, inclusi metodi per la gestione di operatori e centri di monitoraggio.

5.1.3 RemoteServiceImpl

Implementa `RemoteService`, fornendo la logica per l'invocazione dei metodi di accesso al database grazie all'utilizzo dell'istanza DatabaseConnection.

5.1.4 ServerRMI

Avvia il server RMI, gestisce le credenziali per il database e registra l'oggetto remoto nel registro RMI. Questo modulo assicura che i client possano interagire con il sistema back-end di monitoraggio climatico in modo sicuro e efficace.

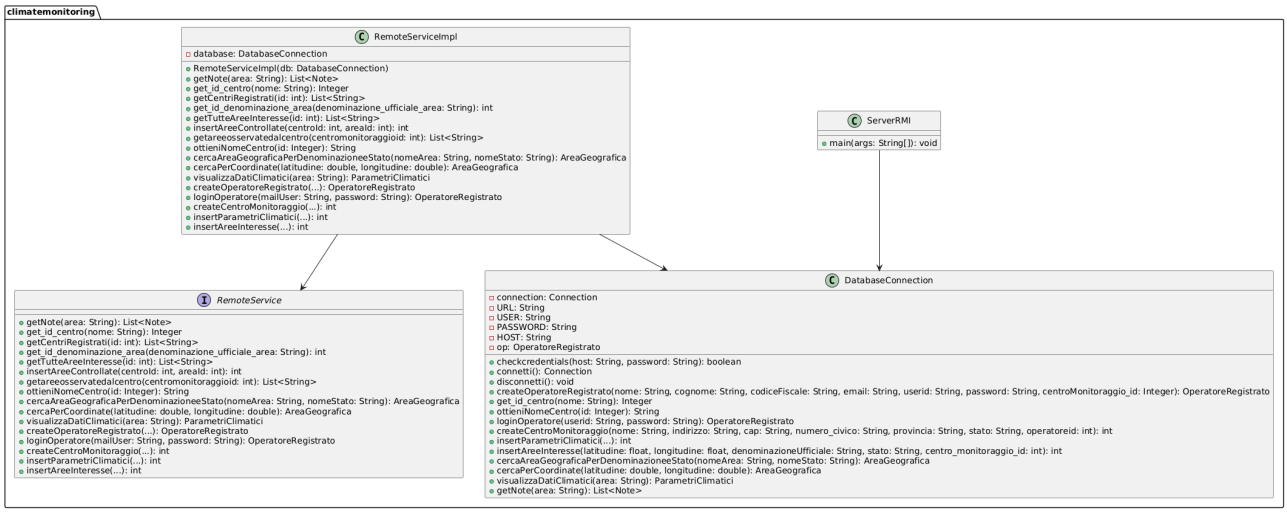

Figure 3: Class Diagram: serverCM , per sapere cosa fa ogni metodo è consigliato visionare la documentazione JavaDoc

5.2 ClientCM

Il modulo `clientCM` attraverso un'interfaccia grafica intuitiva sistema consente agli operatori di registrarsi, effettuare il login e gestire i dati climatici relativi a specifiche aree geografiche. Questo modulo rappresenta idealmente il front-end ed è costituito da diverse classi **frame** , ognuna della quali con una funzione specifica . Le interazioni tra client e server e database avvengono tramite RMI (Remote Method Invocation), permettendo un accesso efficiente e sicuro alle informazioni climatiche mostrate dallo stesso module clientCM.

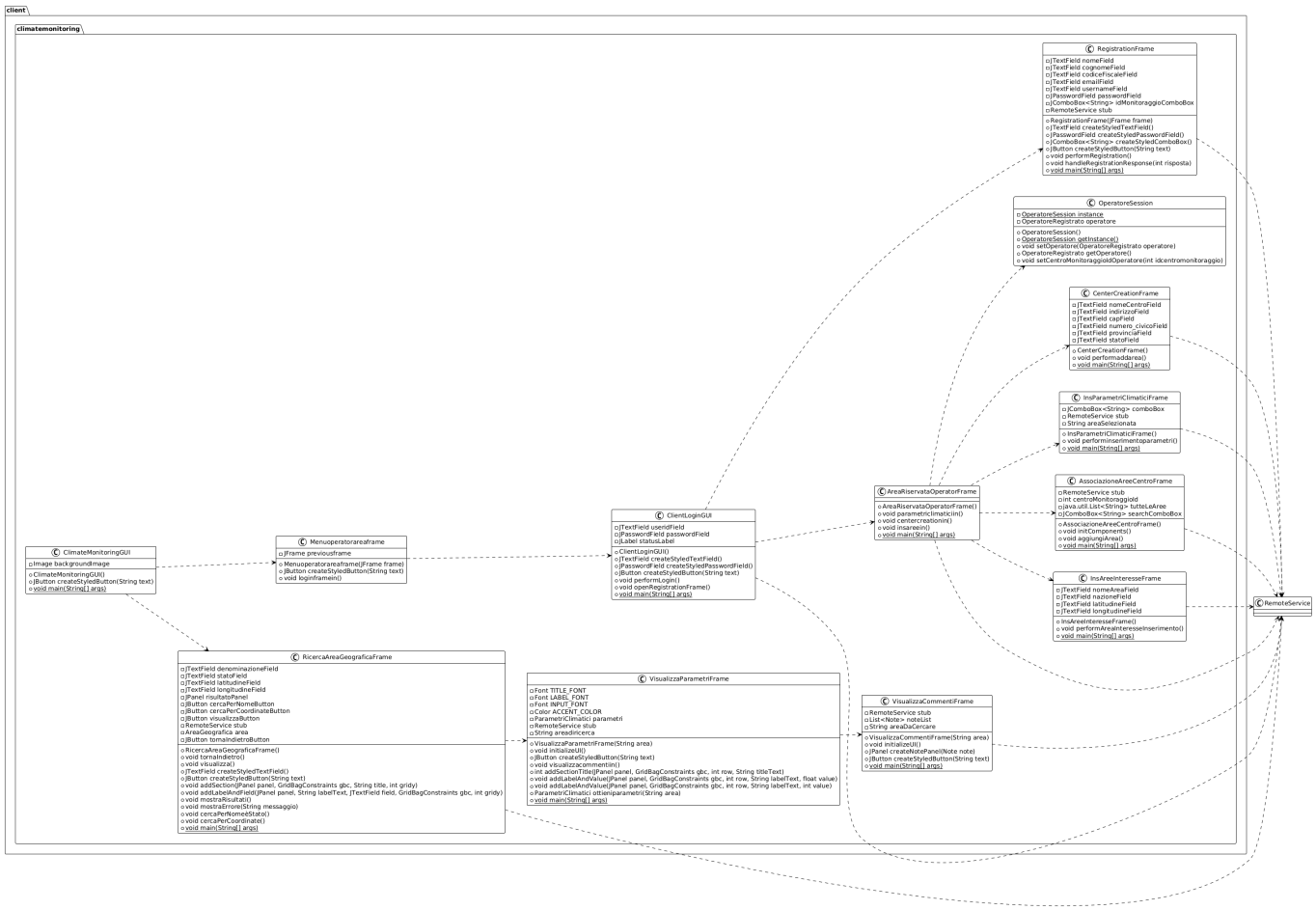

Figure 4: Class diagram ClientCM: per un'idea piu chiara sui metodi visualizzare la JavaDOC

5.2.1 ClientLoginGUI

La classe `ClientLoginGUI` rappresenta l'interfaccia grafica per il login degli operatori. Estendendo `JFrame`, fornisce un modulo dove l'operatore può inserire il proprio ID e password. La validità delle credenziali viene verificata tramite comunicazione remota con il server. Se il login ha successo, l'operatore viene reindirizzato all'area riservata.

5.2.2 ClimateMonitoringGUI

La classe `ClimateMonitoringGUI` fornisce un'interfaccia principale per la gestione dei dati climatici. È responsabile della navigazione tra le diverse funzionalità dell'applicazione, permettendo all'operatore di cercare e visualizzare dati climatici, accedere all'area operatori e uscire dall'applicazione.

5.2.3 AreaRiservataOperatorFrame

La classe `AreaRiservataOperatorFrame` rappresenta la finestra dell'area riservata per gli operatori nel sistema di monitoraggio climatico. Questa finestra offre diverse funzionalità, tra cui la gestione dei parametri di monitoraggio, la creazione di centri di monitoraggio e l'associazione di aree di interesse.

5.2.4 AssociazioneAreeCentroFrame

La classe `AssociazioneAreeCentroFrame` fornisce un'interfaccia grafica attraverso la quale gli operatori possono associare aree esistenti al proprio centro di monitoraggio. Gli operatori possono selezionare un'area da un elenco e aggiungerla al centro, facilitando così la gestione delle aree monitorate.

5.2.5 InsAreeInteresseFrame

La classe `InsAreeInteresseFrame` consente agli operatori di inserire nuove aree di interesse nel sistema. Questa interfaccia permette di registrare dettagli specifici dell'area, che verranno automaticamente inseriti nel database delle aree controllate.

5.2.6 InsParametriClimaticiFrame

La classe `InsParametriClimaticiFrame` gestisce l'inserimento dei parametri climatici nel sistema. Fornisce un'interfaccia per la raccolta di dati climatici, come velocità del vento, umidità e temperatura, consentendo agli operatori di registrare osservazioni dettagliate per le aree monitorate.

5.2.7 CenterCreationFrame

La classe `CenterCreationFrame` rappresenta il frame per la creazione di nuovi centri di monitoraggio. Gli operatori possono inserire informazioni dettagliate riguardanti il centro, come nome, indirizzo e stato, e inviare queste informazioni al server per la registrazione.

5.2.8 RicercaAreaGeograficaFrame

La classe `RicercaAreaGeograficaFrame` gestisce l'interfaccia utente per la ricerca di aree geografiche. Gli operatori possono cercare aree per denominazione o coordinate e visualizzare i risultati delle ricerche, utilizzando RMI per interagire con il database delle aree.

5.2.9 VisualizzaParametriFrame

La classe `VisualizzaParametriFrame` consente di visualizzare i parametri climatici relativi a un'area specifica. Utilizza RMI per recuperare i dati dal server e presenta le informazioni in un formato facilmente leggibile.

5.2.10 VisualizzaCommentiFrame

La classe `VisualizzaCommentiFrame` fornisce un'interfaccia per visualizzare i commenti associati a un'area specifica. Gli operatori possono vedere le note riguardanti le condizioni climatiche e altre osservazioni importanti.

5.2.11 Menuoperatorareaframe

La classe `Menuoperatorareaframe` rappresenta la finestra principale per gli operatori, permettendo loro di accedere a diverse funzioni come login e registrazione. Fornisce anche un'interfaccia per la gestione dei dati climatici.

5.3 Common

Il modulo Common è una parte fondamentale dell'applicazione di monitoraggio climatico. Esso contiene classi essenziali per la gestione di operazioni di base, come la rappresentazione di aree geografiche, la registrazione di note climatiche, la gestione degli operatori e la rappresentazione di parametri climatici. Le classi di questo modulo definiscono i dati serializzati scambiati tramite RMI da database a client . Di seguito sono descritte le classi incluse nel modulo, accompagnate da dettagli sulle loro funzionalità e attributi.

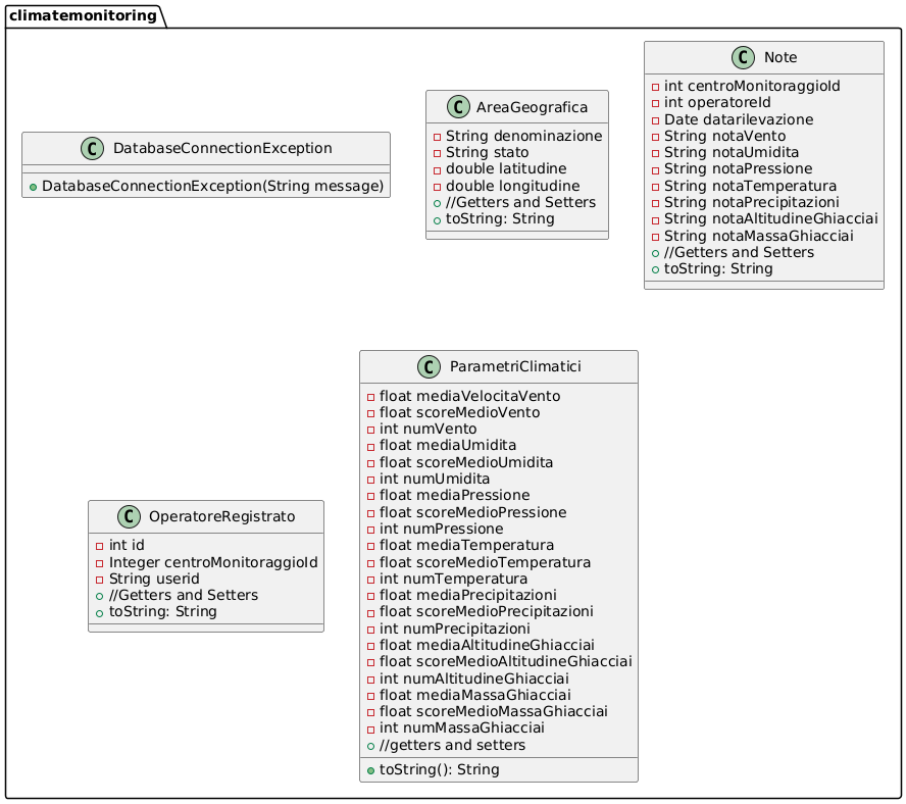

Figure 5: Class Diagram Common

5.3.1 DatabaseConnectionException

La classe `DatabaseConnectionException` estende `Exception` e rappresenta un'eccezione specifica che si verifica durante le operazioni di connessione al database. Essa accetta un messaggio di errore come parametro nel suo costruttore, permettendo di fornire dettagli su ciò che è andato storto durante la connessione.

5.3.2 AreaGeografica

La classe `AreaGeografica` rappresenta un'area geografica con dettagli relativi alla sua posizione in coordinate geografiche , alla sua denominazione ufficiale e Stato di appartenenza. Questa classe implementa l'interfaccia `Serializable`, consentendo la serializzazione degli oggetti. Gli attributi principali includono:

- `denominazione`: rappresenta il nome dell'area geografica.
- `stato`: indica lo stato in cui si trova l'area geografica.
- `latitudine`: memorizza la latitudine dell'area.
- `longitudine`: memorizza la longitudine dell'area.

Il costruttore della classe accetta tutti i parametri sopra menzionati per inizializzare un'istanza di `AreaGeografica`. Inoltre, la classe fornisce metodi getter e setter per ciascun attributo e un metodo `toString()` per restituire una rappresentazione testuale dell'oggetto.

5.3.3 Note

La classe `Note` rappresenta le note registrate dagli operatori per un centro di monitoraggio. Anch'essa implementa l'interfaccia `Serializable` e dispone di vari attributi per memorizzare informazioni climatiche. Gli attributi principali includono:

- `centroMonitoraggioId`: l'ID del centro di monitoraggio che ha registrato la nota.
- `operatoreId`: l'ID dell'operatore che ha inserito la nota.
- `datarilevazione`: la data in cui è stata effettuata la rilevazione.
- `notaVento`, `notaUmidita`, `notaPressione`, `notaTemperatura`, `notaPrecipitazioni`, `notaAltitudineGhiacciai`, `notaMassaGhiacciai`: note specifiche su vari parametri climatici.

Il costruttore della classe richiede tutti i parametri per inizializzare un'istanza di `Note` e fornisce metodi getter e setter per ciascun attributo. Inoltre, è presente un metodo `toString()` che restituisce una rappresentazione testuale della nota.

5.3.4 OperatoreRegistrato

La classe `OperatoreRegistrato` rappresenta un operatore registrato con un ID, un ID del centro di monitoraggio e un ID utente. Anche questa classe implementa `Serializable`. Gli attributi principali includono:

- `id`: l'ID dell'operatore.
- `centroMonitoraggioId`: l'ID del centro di monitoraggio associato all'operatore (può essere `null`).

- `userid`: l'ID utente dell'operatore.

La classe fornisce vari costruttori, metodi getter e setter per ciascun attributo, permettendo di gestire facilmente le informazioni relative agli operatori.

5.3.5 ParametriClimatici

La classe `ParametriClimatici` rappresenta i parametri climatici registrati. Essa è serializzata e contiene variabili per memorizzare i dati climatici principali. Gli attributi principali includono:

- `mediaVelocitaVento`, `scoreMedioVento`, `numVento`: dati relativi alla velocità del vento.
- `mediaUmidita`, `scoreMedioUmidita`, `numUmidita`: dati relativi all'umidità.
- `mediaPressione`, `scoreMedioPressione`, `numPressione`: dati relativi alla pressione atmosferica.
- `mediaTemperatura`, `scoreMedioTemperatura`, `numTemperatura`: dati relativi alla temperatura.
- `mediaPrecipitazioni`, `scoreMedioPrecipitazioni`, `numPrecipitazioni`: dati relativi alle precipitazioni.
- `mediaAltitudineGhiacciai`, `scoreMedioAltitudineGhiacciai`, `numAltitudineGhiacciai`: dati relativi all'altitudine dei ghiacciai.
- `mediaMassaGhiacciai`, `scoreMedioMassaGhiacciai`, `numMassaGhiacciai`: dati relativi alla massa dei ghiacciai.

Il costruttore della classe accetta tutti i parametri sopra menzionati per inizializzare un'istanza di `ParametriClimatici`. La classe fornisce anche metodi getter per ciascun attributo e un metodo `toString()` per restituire una rappresentazione dettagliata dei parametri climatici.

In sintesi, il modulo Common fornisce una base solida per la gestione delle informazioni climatiche e degli operatori, facilitando l'interazione con i dati e la loro registrazione.

Di seguito è possibile vedere un' immagine riassuntiva di questi 3 moduli e in particolari della relazione di dipendenza tra le classi dei 3 moduli. Le classi del modulo ClientCM dipendono dalle classi del module Common per rappresentare i dati da far visualizzare all'utente o per definirli secondo gli input dell'utente. Le classi del modulo ClientCM dipendono dal servizio remoto di ServerCM per avere una comunicazione con il database . Anche Remote Service di ServerCM utilizza le classi di Common per rappresentare i dati necessari per la comunicazione .

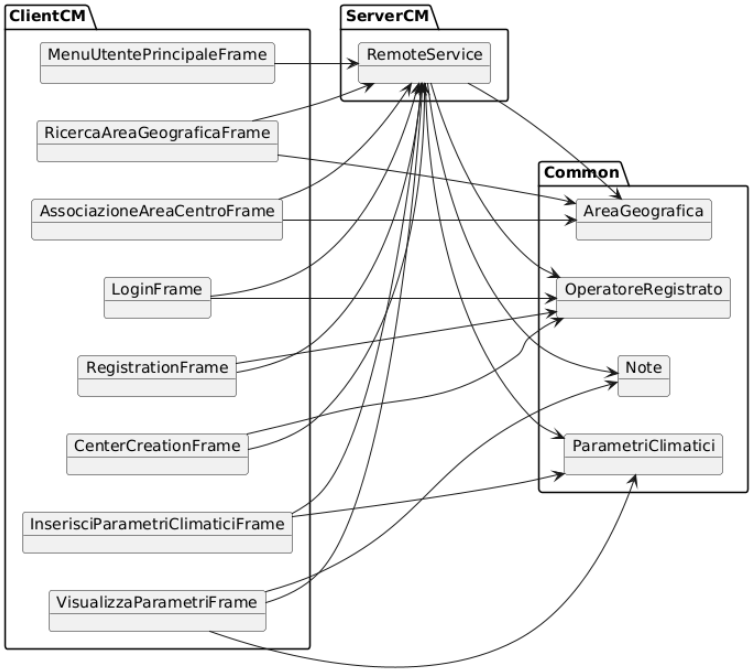

Figure 6: Object Diagram

5.4 Database PostgreSql : SERVERLABB

Il database è distribuito su pgAdmin 4, servizio di PostgreSql. Al database l'admin accede tramite alcune credenziali come password , nome utente , nome host e nome database tramite il server . Quest'ultimo proverà a definire una connessione con il database secondo questi parametri tramite la classe dedicata DatabaseConnection.

5.5 Analisi dei requisiti

L'analisi dei requisiti è il primo passo fondamentale nella progettazione di un sistema informativo, in quanto consente di comprendere le necessità degli utenti finali e di tradurle in specifiche tecniche. Nel nostro caso, i requisiti riguardano un sistema per il monitoraggio di parametri climatici in diverse aree geografiche, gestito da vari centri di monitoraggio con operatori registrati.

I requisiti funzionali principali del sistema sono i seguenti:

- **Monitoraggio delle Aree**: Il sistema deve consentire la registrazione e il monitoraggio dei parametri climatici in diverse aree di interesse.
- **Gestione dei Centri di Monitoraggio**: Ogni centro di monitoraggio ha la responsabilità di monitorare specifiche aree e di registrare i dati relativi ai parametri climatici riguardo una specifica area.
- **Registrazione degli Operatori**: Gli operatori devono essere registrati nel sistema e associati a un centro di monitoraggio (opzionalmente) , con la possibilità di raccogliere dati sui parametri

climatici.

- **Rilevazione dei Parametri Climatici**: I parametri da monitorare includono temperatura, umidità, pressione , velocità del vento , altezza e massa ghiacciai . Quest'ultimi devono essere monitorati attraverso uno score da 1 a 5 , un valore numerico che indichi l'intensità di questo parametro nella sua unità di misura e infine una nota alfanumerica opzionale . Ogni operatore può registrare più misurazioni in diverse aree. NOTA: la tabella del DB OperatoriRegistrati deve essere aggiornata con un riferimento al centro di monitoraggio appena creato, che risulterà essere il centro di riferimento dell'operatore.

In aggiunta, i requisiti non funzionali includono la necessità di garantire l'integrità dei dati attraverso vincoli di unicità di alcuni dati come il codice fiscale di un operatore , la gestione di relazioni tra entità e valori di score dei parametri climatici compresi tra 1 e 5.

5.6 Concettualizzazione dello Schema ER e possibile traduzione logica

A partire dall'analisi dei requisiti, è stato progettato uno schema Entità-Relazione (ER) che riflette le entità e le relazioni principali del sistema:

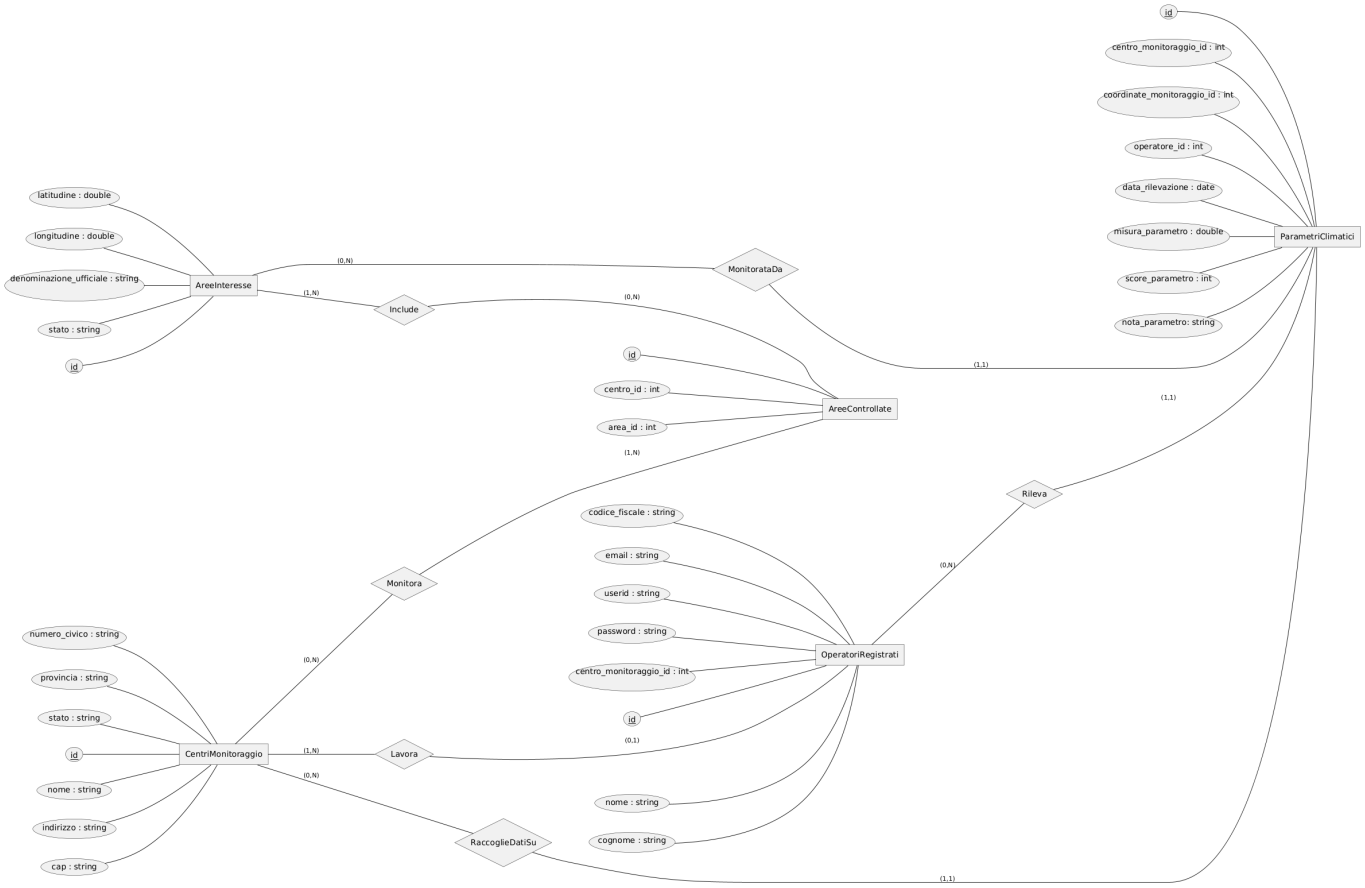

Figure 7: Diagramma ER:per praticità gli attributi di parametri sono stati semplificati a **attributo-parametro**. I parametri sono rispettivamente: vento, umidità, pressione, temperatura, precipitazione, altitudine dei ghiacciai, massa dei ghiacciai.

5.6.1 AreeInteresse

Memorizza i luoghi geografici oggetto di monitoraggio.

Attributi: id, latitudine, longitudine, denominazione ufficiale, stato. L'attributo **id** è vincolo di identificazione , puo' essere tradotta logicamente in una **primary key serial** , l' attributo aumenta di una unità ad ogni nuova creazione di questa entità mentre denominazione ufficiale è idealmente **unique** perchè non è possibile inserire un'area già esistente nel database.
Associazioni: Include (1-N, un'area può essere inclusa in più aree controllate , dunque puo' esser controllata da piu' centri . Inoltre se un 'area è stata creata da un utente allora quest'ultima per forza appartiene ad almeno un centro), MonitorataDa (0-N, un'area può avere zero o molte rilevazioni di parametri associate).

5.6.2 CentriMonitoraggio

Entità che rappresenta il centro di monitoraggio che da specifiche monitora aree e gestisce parametri climatici.

Attributi: id, nome, indirizzo, numero civico, provincia, stato, cap. L'attributo **id** è vincolo di identificazione , puo' essere tradotta logicamente in una **primary key serial** , l' attributo aumenta di una unità ad ogni nuova creazione di questa entità. Gli attributi indirizzo,cap,numero civico devono formare un vincolo unique affinchè un centro di monitoraggio non venga registrato 2 volte.
Associazioni: Monitora (0-N, un centro monitora piu' aree o puo' anche non controllarle quando quest'ultimo viene creato), Lavora (1-N, un centro ha piu' operatori dipendenti , ne ha per forza uno che colui che crea il suddetto centro), RaccoglieDatiSu (0-N, un centro raccoglie dati su diversi parametri climatici ma puo' anche non averne raccolti).

5.6.3 OperatoriRegistrati

L'entità che rappresenta l'operatore registrato memorizzando le sue informazioni .

Attributi: id, nome, cognome, codice fiscale, email, userid, password, centro monitoraggio id.L'attributo **id** è vincolo di identificazione , puo' essere tradotta logicamente in una **primary key serial** , l' attributo aumenta di una unità ad ogni nuova creazione di questa entità . Userid e codice fiscale possono essere vincoli unique . Idealmente è inverosimile avere utenti con codice fiscale e user id , per natura univoci , identici . Centro monitoraggio id deve essere chiave esterna di CentriMonitoraggio (id), questo perchè ogni utente registrato puo' avere un centro di monitoraggio di afferenza . Puo' essere adottata l'opzione **ON DELETE SET NULL** per mantenere l'integrità referenziale nei database, impostando il valore della chiave esterna a NULL quando la riga correlata nella tabella primaria viene eliminata, consentendo di conservare il record senza un riferimento non valido. Questo valore centro monitoraggio id non è obbligatorio infatti viene inzializzato a null qualora l'utente non abbia un centro associato.
Associazioni: Lavora (0-1, ogni operatore lavora in un solo centro ma puo' anche non avere un centro in cui lavori), Rileva (0-N, un operatore può effettuare molte rilevazioni come non effettuarle).

5.6.4 AreeControllate

Questa entità rappresenta idealmente un **rapporto di monitoraggio** tra un centro e un' area di interesse . Questa entità , anche se non specificatamente richiesta nelle specifiche , è stata pensata per l' operazione di inserimento dei parametri climatici affinchè un utente appartenente al centro di monitoraggio x possa inserire parametri solo ed esclusivamente per le aree y del suo centro x. Inoltre è stata pensata per rispettare il requisito : " **creare centri di monitoraggio con l'elenco delle aree di interesse**" : l'entità favorisce un mapping diretto tra il centro x e le n-aree di interesse .

Attributi: id, centro id, area id. L'attributo **id** è vincolo di identificazione , puo' essere tradotta logicamente in una **primary key serial** , l' attributo aumenta di una unità ad ogni nuova creazione

di questa entità . L'attributo area id puo' essere chiave esterna di areeinteresse(id) mentre l'attributo centro id puo' essere chiave esterna di centrimonitoraggio(id) , entrambi con l'opzione **ON DELETE CASCADE** il quale garantisce l'integrità referenziale eliminando automaticamente i record figli (nelle tabelle con chiavi esterne) quando il record correlato nella tabella primaria viene eliminato. Dunque è stato considerato il fatto che se il centro x viene eliminato è ovvio implicare che vengano eliminate anche le sue associazioni di monitoraggio con l' area y , e viceversa.
Associazioni: Include (1-1, relazione univoca tra un'area di interesse e un centro monitoraggio). Monitora(1,N) perchè un area puo' essere monitorata da uno o piu' centri,

5.6.5 ParametriClimatici

Entità che rappresenta l'insieme di parametri climatici relativa ad un' area di uno specifico centro in una data specifica che memorizza i diversi valori per ogni parametro .

Attributi: id, centro monitoraggio id, coordinate monitoraggio id, operatore id,data rilevazione, velocià vento, score vento , nota vento , misura umidità , score umidità , nota umidità , misura pressione dell'aria , score pressione dell'aria , nota pressione dell'aria , misura temperatura , score temperatura , note temperatura , misura precipitazioni , score precipitazioni , note precipitazione , misura massa ghiacciai , score massa ghiacciai , note massa ghiacciai , altezza ghiacciai , score ghiacciai , note altezza ghiacciai . L'attributo **id** è vincolo di identificazione , puo' essere tradotta logicamente in una **primary key serial** , l' attributo aumenta di una unità ad ogni nuova creazione di questa entità . Gli attributi coordinate monitoraggio id , centro monitoraggio id , operatore id devono essere chiavi esterne rispettivamente di areeinteresse , centrimonitoraggio , operatoriregistrati con opzione **ON DELETE RESTRICT** ,utilizzata per impedire l'eliminazione nelle tabelle padre se esistono record nella tabella figlia che fanno riferimento ad esso. Questo perchè se una rilevazione è stata eseguita su un' area risulta verosimile pensare che questa rilevazioni persista nel tempo a prescindere dal centro , dall'operatore o dalla stessa area . Sui campi score parametro deve essere definito il vincolo **CHECK (parametro) BETWEEN 1 AND 5** che implica che i valori di dominio siano x appartenente a 1,2,3,4,5 come secondo specifiche .
Associazioni: Rileva (1-1), una rilevazione di parametri è eseguita da un operatore specifico , MonitorataDa (1-1) , ogni rilevazione è associata ad una specifica area di interesse e RaccoglieDatiSu(1-1) , una rilevazione è associata ad uno ed un solo centro.

Tutti i campi devono essere non nulli , eccetto il campo **centro monitoraggio id** di **OperatoriRegistrati** e i campi **note parametro** di **ParametriClimatici** .
Il requisito : **la tabella del DB OperatoriRegistrati deve essere aggiornata con un riferimento al centro di monitoraggio appena creato, che risulterà essere il centro di riferimento dell'operatore** sarà garantita grazie alla query di **Update** SQL 9) a pagina 24.

5.6.6 Schema ER ristrutturato

Lo schema ER proposto non è stato ristrutturato ulteriormente poiché è già progettato in modo ottimale, rispettando i principi di normalizzazione e i requisiti progettuali. Di seguito le motivazioni

principali:

- **Normalizzazione avanzata:** Lo schema è normalizzato fino alla terza forma normale (3NF), garantendo una chiara separazione delle responsabilità tra le entità ed evitando ridondanze come attributi composti. Ogni entità contiene solo attributi semplici.
- **Relazioni chiare e ben definite:** Le associazioni (*Lavora*, *Monitora*, *Include*, ecc.) sono state modellate con vincoli di cardinalità precisi e referenzialità esplicite, riducendo la necessità di ulteriori tabelle intermedie. Inltre non vi sono attributi di associazioni.
- **Rappresentazione delle dipendenze funzionali:** Le chiavi primarie e i vincoli (*UNIQUE*, *ON DELETE CASCADE*, *CHECK*) mantengono l'integrità e la consistenza dei dati, evitando conflitti o inconsistenze tra le entità.

5.7 Altri vincoli

Altri vincoli sui campi delle entità del database sono stati definiti in seguito dalle varie classi ClientCM che controllano i dati forniti dall'utente , per esempio si verifica che si inseriscano email che contengano l'elemento @ o che il codice fiscale di utente abbia 16 cifre quando un utente si registra , si verifica che l'area inserita dell'utente abbia latitutine e longitudine consistenti , si verifica che quando l'utente inserisce i parametri inserisca date di rilevazioni corrette o che alcuni campi come per esempio la velocità abbiano valori consistenti (per esempio valori superiori allo 0 , la velocità non puo' esser negativa). Di seguito viene mostrato un estratto di un controllo della classe InsAreeInteresseFrame :

```
    // Controlla se la data    nel formato corretto YYYY-MM-DD
            if (!dataRilevazione.matches("\\d{4}-\\d{2}-\\d{2}")) {
                // Mostra un messaggio di errore se il formato    errato
                JOptionPane.showMessageDialog(this, "Errore: La data deve
                    essere nel formato YYYY-MM-DD. Inserisci una data valida."
                    , "Errore formato data", JOptionPane.ERROR_MESSAGE);
                return; // Blocca il processo di inserimento
            }
//controlla se la data non esiste del tipo : 2022-35-36
            try {
                Date sqlDate = Date.valueOf(dataRilevazione);
            } catch (IllegalArgumentException ex) {
                JOptionPane.showMessageDialog(this, "Errore: Data non valida."
                    , "Errore", JOptionPane.ERROR_MESSAGE);
                return;
            }
```

5.8 Traduzione a schema relazionale : script sql

Qua di seguito sono visualizzabili gli script sql utilizzati per produrre le tabelle del database e le varie operazione di query come visualizzazione , ricerca , inserimento , modifica. Nella sezione predecente sono stati discusse le motivazioni dei vincoli applicati ai vari campi.

```
CREATE TABLE areeinteresse (
    id serial PRIMARY KEY,
    latitudine double precision NOT NULL,
    longitudine double precision NOT NULL,
    denominazione_ufficiale varchar(100) NOT NULL UNIQUE,
    stato varchar(100) NOT NULL
);
```

Tabella centrimonitoraggio

```
CREATE TABLE centrimonitoraggio (
    id serial PRIMARY KEY,
    nome varchar NOT NULL,
    indirizzo varchar NOT NULL,
    cap varchar NOT NULL,
    numero_civico varchar NOT NULL,
    provincia varchar NOT NULL,
    stato varchar NOT NULL,
    UNIQUE (indirizzo, cap, numero_civico)
);
```

Tabella areecontrollate

```
CREATE TABLE areecontrollate (
    id serial PRIMARY KEY,
    centro_id integer NOT NULL,
    area_id integer NOT NULL,
    UNIQUE (centro_id, area_id),
    FOREIGN KEY (area_id) REFERENCES areeinteresse(id) ON DELETE CASCADE,
    FOREIGN KEY (centro_id) REFERENCES centrimonitoraggio(id) ON DELETE CASCADE
);
```

Tabella operatoriregistrati

```
CREATE TABLE operatoriregistrati (
    id serial PRIMARY KEY,
    nome varchar(100) NOT NULL,
    cognome varchar(100) NOT NULL,
    codice_fiscale varchar(16) NOT NULL UNIQUE,
    email varchar(100) NOT NULL,
    userid varchar(50) NOT NULL UNIQUE,
    password varchar(100) NOT NULL,
    centro_monitoraggio_id integer,
    FOREIGN KEY (centro_monitoraggio_id)
```

```
        REFERENCES centrimonitoraggio(id) ON DELETE SET NULL
);
```

Tabella parametriclimatici

```
CREATE TABLE parametriclimatici (
    id serial PRIMARY KEY,
    centro_monitoraggio_id integer NOT NULL,
    coordinate_monitoraggio_id integer NOT NULL,
    operatore_id integer NOT NULL,
    data_rilevazione date NOT NULL,

    velocita_vento double precision NOT NULL,
    score_vento integer NOT NULL,
    nota_vento text,

    umidita double precision NOT NULL,
    score_umidita integer NOT NULL
    CHECK (score_umidita BETWEEN 1 AND 5),
    nota_umidita text,

    pressione double precision NOT NULL,
    score_pressione integer NOT NULL
    CHECK (score_pressione BETWEEN 1 AND 5),
    nota_pressione text,

    temperatura double precision NOT NULL,
    score_temperatura integer NOT NULL
    CHECK (score_temperatura BETWEEN 1 AND 5),
    nota_temperatura text,

    precipitazioni double precision NOT NULL,
    score_precipitazioni integer NOT NULL
    CHECK (score_precipitazioni BETWEEN 1 AND 5),
    nota_precipitazioni text,

    altitudine_ghiacciai double precision NOT NULL,
    score_altitudine_ghiacciai integer NOT NULL
    CHECK(score_altitudine_ghiacciai BETWEEN 1 AND 5),
    nota_altitudine_ghiacciai text,

    massa_ghiacciai double precision NOT NULL,
    score_massa_ghiacciai integer NOT NULL
```

```
    CHECK (score_massa_ghiacciai BETWEEN 1 AND 5),
    nota_massa_ghiacciai text,

    FOREIGN KEY (coordinate_monitoraggio_id) REFERENCES areeinteresse(id),
    FOREIGN KEY (centro_monitoraggio_id) REFERENCES centrimonitoraggio(id),
    FOREIGN KEY (operatore_id) REFERENCES operatoriregistrati(id)
);
```

Query SQL

```
1. Creazione di un Operatore Registrato
INSERT INTO OperatoriRegistrati (nome, cognome, codice_fiscale,
email, userid, password, centro_monitoraggio_id)
VALUES (?, ?, ?, ?, ?, ?, ?)

2. Ottenere l'ID del Centro di Monitoraggio per nome
SELECT id FROM centrimonitoraggio WHERE nome ILIKE ?

3. Ottenere il nome del Centro per ID
SELECT * FROM CentriMonitoraggio WHERE id = ?

4. Verifica se un Operatore esiste
SELECT COUNT(*) FROM OperatoriRegistrati WHERE id=?

5. Verifica se un Centro di Monitoraggio esiste
SELECT COUNT(*) FROM CentriMonitoraggio WHERE id = ?

6. Controllo di esistenza di un record con Codice Fiscale o UserID
SELECT COUNT(*) FROM OperatoriRegistrati
WHERE codice_fiscale ILIKE ? OR userid ILIKE ?

7. Login di un Operatore
SELECT id, centro_monitoraggio_id FROM OperatoriRegistrati
WHERE userid = ? AND password = ?

8. Creazione di un Centro di Monitoraggio
INSERT INTO CentriMonitoraggio (nome, indirizzo,
cap, numero_civico, provincia, stato)
VALUES (?, ?, ?, ?, ?, ?)

9. Aggiornamento dell'Operatore con ID del Centro di Monitoraggio
UPDATE OperatoriRegistrati SET centro_monitoraggio_id = ? WHERE id = ?
```

10. Inserimento Parametri Climatici

```
INSERT INTO parametriclimatici (centro_monitoraggio_id, coordinate_monitoraggio_id,
operatore_id, data_rilevazione,
velocita_vento, score_vento, nota_vento,
umidita, score_umidita, nota_umidita,
pressione, score_pressione, nota_pressione,
temperatura, score_temperatura, nota_temperatura,
precipitazioni, score_precipitazioni,
nota_precipitazioni, altitudine_ghiacciai,
score_altitudine_ghiacciai,
nota_altitudine_ghiacciai, massa_ghiacciai,
score_massa_ghiacciai, nota_massa_ghiacciai) VALUES (?, ?, ?, ?, ?, ?, ?,
?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

11. Ottenere ID della Denominazione Ufficiale dell'area

```
SELECT id FROM areeinteresse WHERE denominazione_ufficiale = ?
```

12. Verifica che l'Operatore lavori per un Centro di Monitoraggio

```
SELECT COUNT(*) FROM operatoriregistrati WHERE id = ? AND centro_monitoraggio_id = ?
```

13. Inserimento in AreeInteresse

```
INSERT INTO AreeInteresse (latitudine, longitudine, denominazione_ufficiale, stato)
VALUES (?, ?, ?, ?)
```

14. Inserimento in AreeControllate

```
INSERT INTO AreeControllate (centro_id, area_id) VALUES (?, ?)
```

15. Cerca un'area geografica per denominazione ufficiale e stato

```
SELECT * FROM areeinteresse WHERE denominazione_ufficiale ILIKE ? AND stato ILIKE ?
```

16. Cerca un'area geografica per coordinate

```
SELECT * FROM areeinteresse ORDER BY POWER(latitudine - ?, 2) + POWER(longitudine - ?, 2) LIMIT 1
```

17. Visualizza informazioni climatiche di un'area di interesse

```
SELECT STRING_AGG(TO_CHAR(data_rilevazione, 'YYYY-MM-DD'), '; ') AS date_rilevazioni,
AVG(velocita_vento) AS media_velocita_vento, AVG(score_vento) AS score_medio_vento,
COUNT(velocita_vento) AS num_vento,
AVG(umidita) AS media_umidita,
AVG(score_umidita) AS score_medio_umidita,
COUNT(umidita) ASnum_umidita,
AVG(pressione) AS media_pressione,
```

```
AVG(score_pressione) AS score_medio_pressione,
COUNT(pressione) AS num_pressione,
AVG(temperatura) AS media_temperatura,
AVG(score_temperatura) AS score_medio_temperatura,
COUNT(temperatura) AS num_temperatura,
AVG(precipitazioni) AS media_precipitazioni,
AVG(score_precipitazioni) AS score_medio_precipitazioni,
COUNT(precipitazioni) AS num_precipitazioni,
AVG(altitudine_ghiacciai) AS media_altitudine_ghiacciai, AVG(score_altitudine_ghiacciai)
AS score_medio_altitudine_ghiacciai,
COUNT(altitudine_ghiacciai) AS num_altitudine_ghiacciai, AVG(massa_ghiacciai)
AS media_massa_ghiacciai,
AVG(score_massa_ghiacciai)
AS score_medio_massa_ghiacciai,
COUNT(massa_ghiacciai) AS num_massa_ghiacciai
FROM ParametriClimatici
WHERE coordinate_monitoraggio_id = ?

18. Ottenere aree osservate da un centro di monitoraggio
SELECT DISTINCT ai.denominazione_ufficiale
FROM areeinteresse ai JOIN areecontrollate ac
ON ai.id = ac.area_id WHERE ac.centro_id = ?

19. Ottenere tutte le aree di interesse
SELECT DISTINCT denominazione_ufficiale FROM areeinteresse WHERE id > ?

20. Ottenere tutti i centri registrati
SELECT DISTINCT nome FROM centrimonitoraggio WHERE id > ?

21. Ottenere le note associate a un'area di interesse
SELECT centro_monitoraggio_id, operatore_id, data_rilevazione, nota_vento, nota_umidita,
nota_pressione, nota_temperatura, nota_precipitazioni, nota_altitudine_ghiacciai,
nota_massa_ghiacciai FROM public.parametriclimatici WHERE coordinate_monitoraggio_id = ?
```

6 Struttura dinamica del progetto

Analizziamo il comportamento del progetto tramite l'utilizzo di diagrammi UML dinamici:

6.1 State Diagram

Il diagramma rappresenta le possibili transizioni delle finestre del moodule clientCM in risposta a scelte dell'utente , fruitore del progetto ClimateMonitoring.

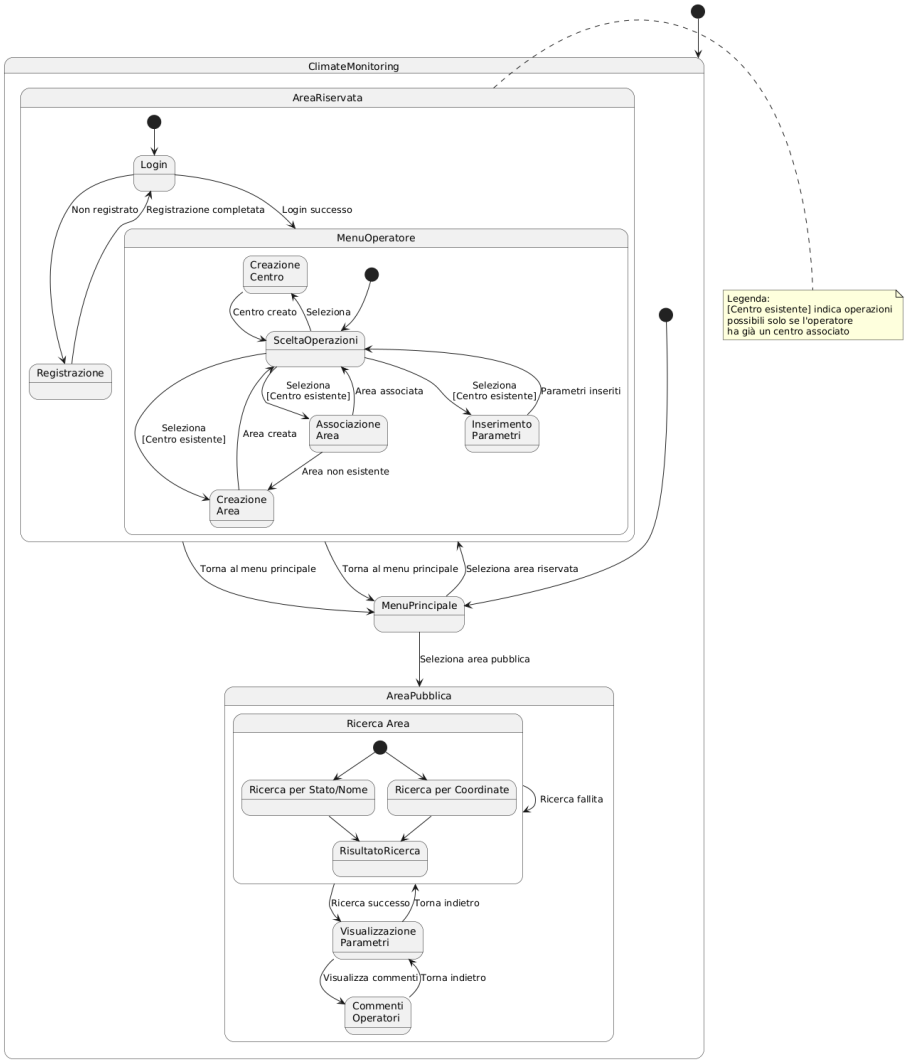

Figure 8: State Diagram

6.2 Sequence Diagram

Il diagramma mostra le interazioni temporali tra oggetti attraverso lo scambio di messaggi. Evidenzia l'ordine cronologico dei messaggi per rappresentare uno scenario specifico.

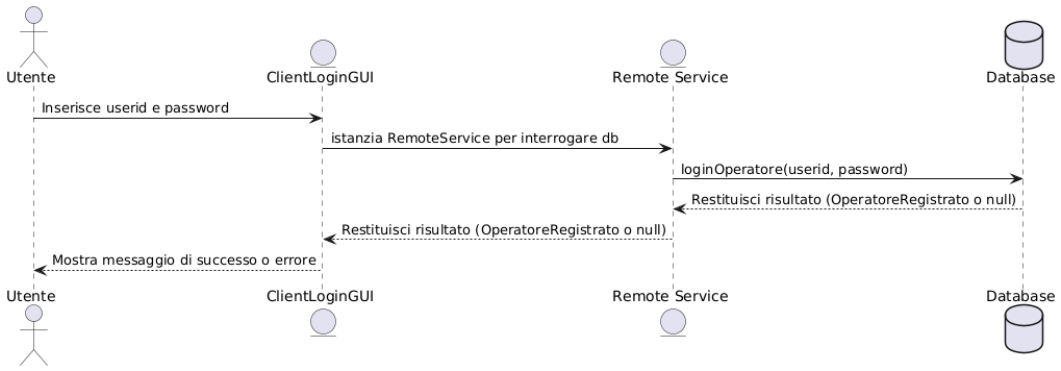

Figure 9: Sequence Diagram: login utente

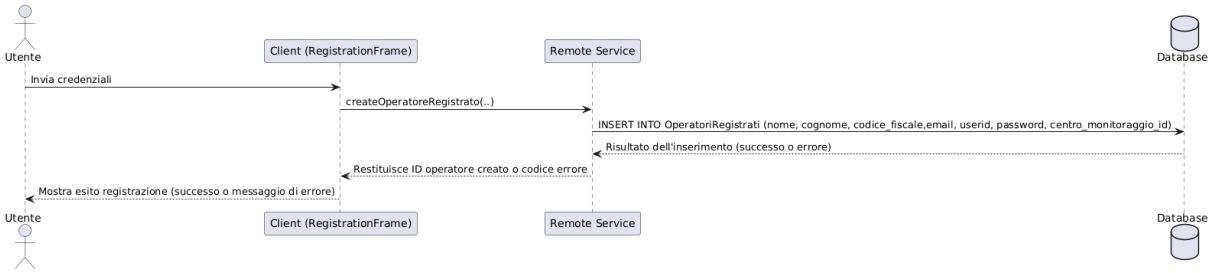

Figure 10: Sequence Diagram: registrazione utente

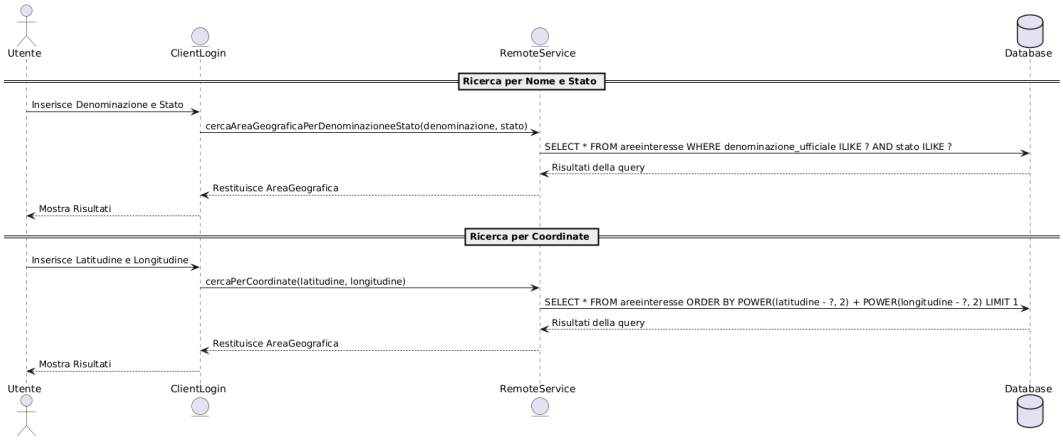

Figure 12: Sequence Diagram: ricerca area di interesse

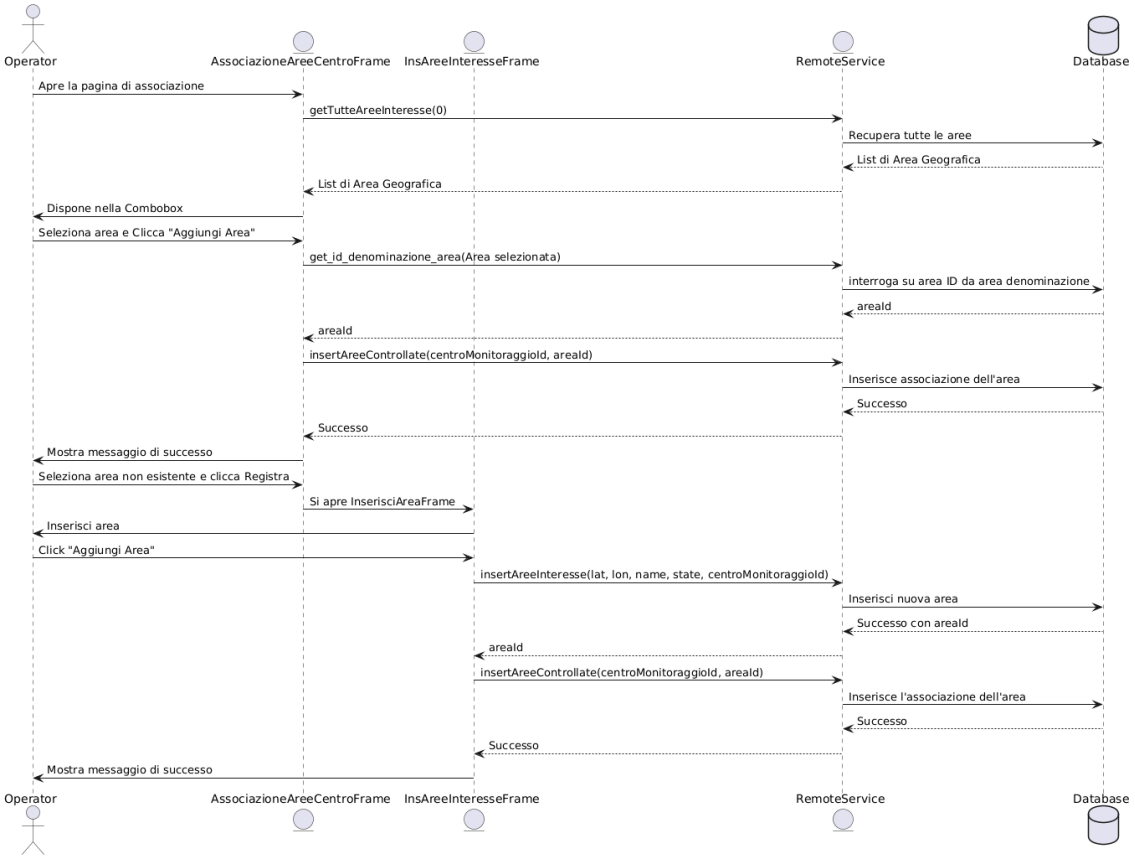

Figure 11: Sequence Diagram: associazione area o eventuale inserimento

6.3 Activity Diagram

Il diagramma che descrive il flusso operativo e il comportamento complessivo del sistema.

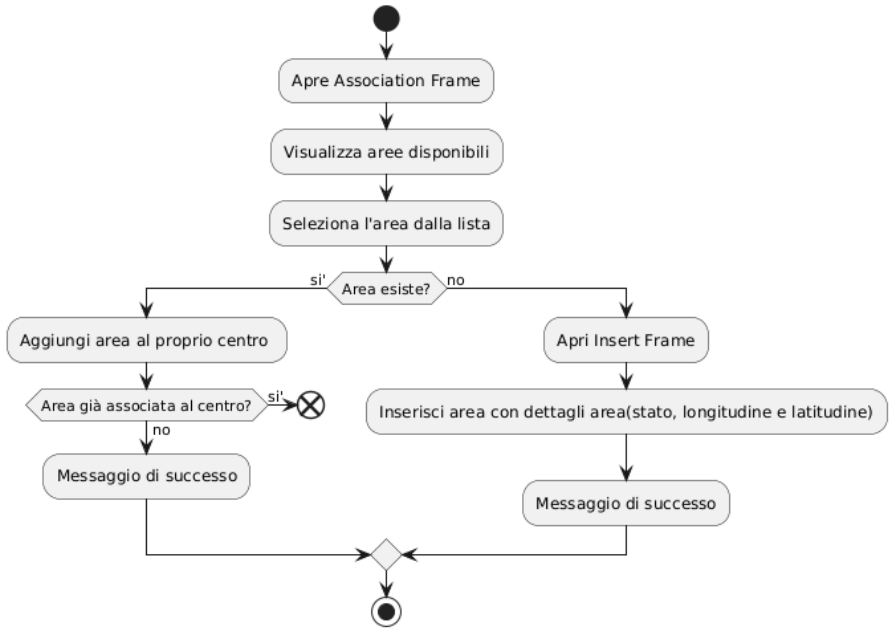

Figure 13: Activity Diagram: associazione ed eventuale inserimento area interesse

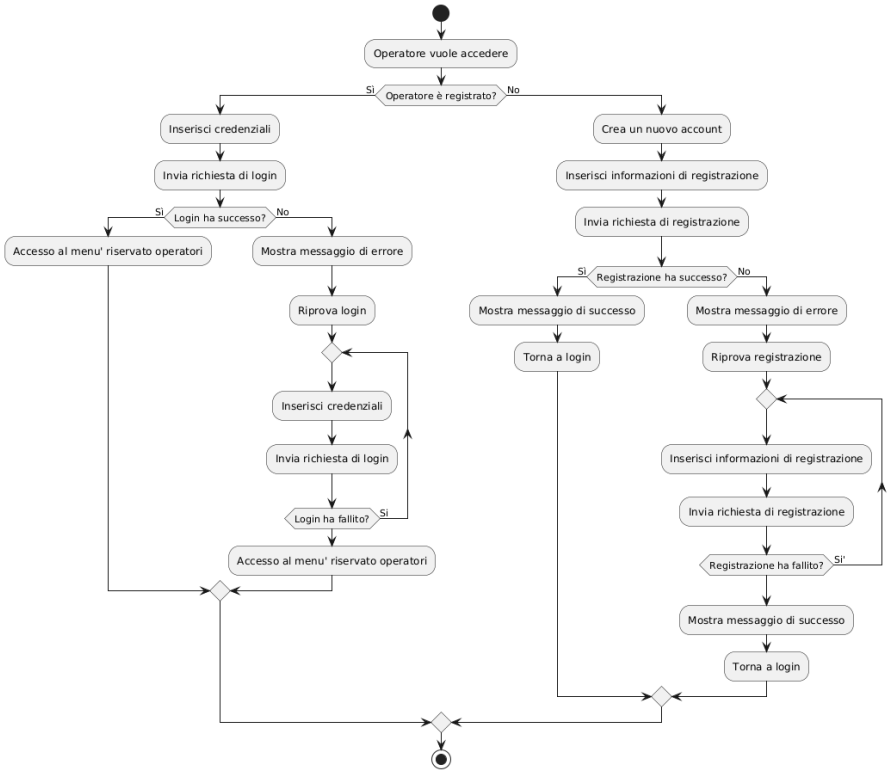

Figure 14: Activity Diagram : login e registrazione operatore

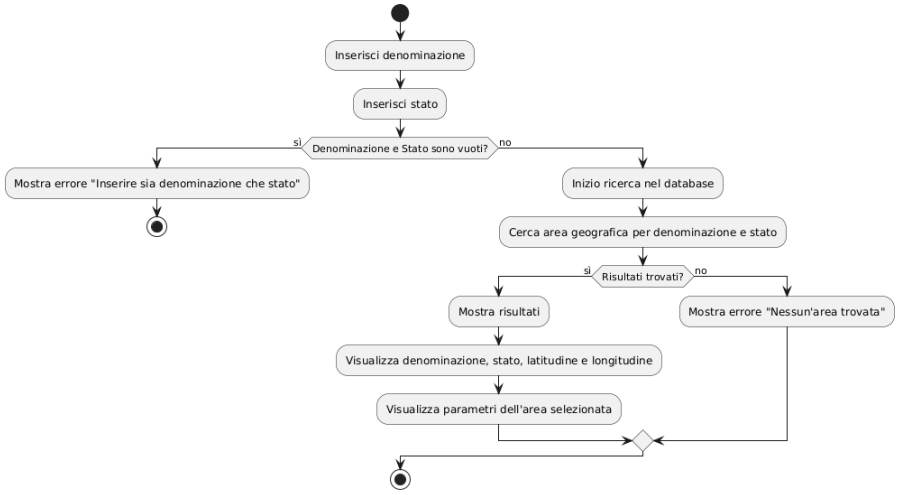

Figure 15: Activity Diagram: ricerca area di interesse per denominazione e stato

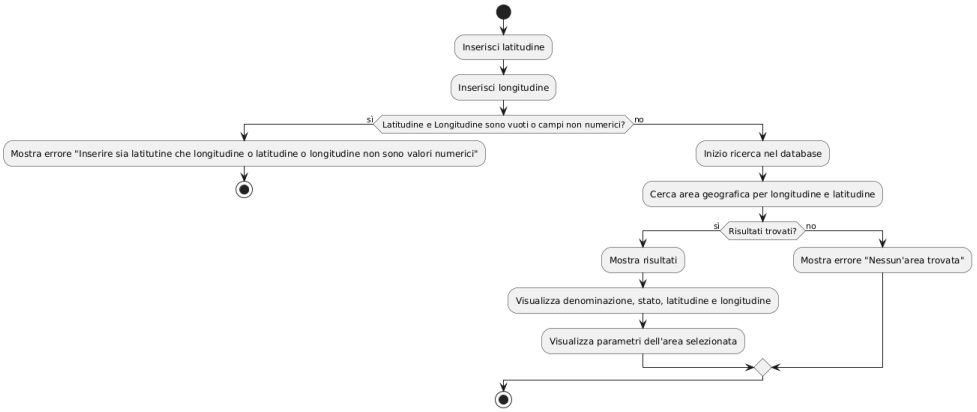

Figure 16: Activity Diagram: ricerca area per coordinate

7 Pattern architetturali

7.1 Model-View-Control (MVC)

Il pattern MVC suddivide un'applicazione in tre componenti principali:

- **Model:** Rappresenta la logica di business e i dati del sistema. In questo caso, le entità come `AreaGeografica`, `Note`, `CentriMonitoraggio`, e `OperatoreRegistrato` appartengono al modello e risiedono nel database.
- **View:** Gestisce l'interfaccia utente e la visualizzazione. La classe `RicercaAreaFrame` funge da finestra grafica e mostra all'utente i dati relativi alle `AreeInteresse`.
- **Controller:** Coordina le interazioni tra View e Model. In questo caso, un oggetto remoto (accessibile tramite RMI) agisce come interfaccia per il `DatabaseConnection`, permettendo alla View di inviare richieste e ricevere risposte relative alle entità.

Esempio: L'utente interagisce con la `RicercaAreaFrame` (**View**), che utilizza l'oggetto remoto (**Controller**) per ottenere i dati dell'`AreaInteresse` (**Model**). Il Controller interroga il database tramite RMI e restituisce il risultato alla View, che lo presenta all'utente.

7.2 Pattern Singleton

Implementazione del Pattern Singleton nella Classe `OperatoreSession`

Il **pattern Singleton** è un design pattern creazionale che garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a tale istanza. Nella classe `OperatoreSession`, il Singleton viene utilizzato per assicurarsi che ci sia un'unica sessione attiva per l'operatore registrato. Questo approccio è utile per mantenere informazioni di sessione consistenti in tutto il ciclo di vita dell'applicazione.

Di seguito è riportato il codice Java che implementa il pattern Singleton:

```
1  /**
2   * Classe che rappresenta una sessione per un operatore registrato.
3   * Implementa il pattern Singleton per garantire che ci sia solo un'istanza di
        OperatoreSession.
4   * @author Moranzoni Samuele
5   * @author Di Tullio Edoardo
6   */
7  public class OperatoreSession {
8      private static OperatoreSession instance;  // Istanza unica della classe
9      private OperatoreRegistrato operatore;     // Riferimento all'operatore
           registrato
10
11     /**
12      * Costruttore privato per impedire la creazione di nuove istanze dall'
            esterno.
13      */
```

```
14      private OperatoreSession() {}
15
16      /**
17       * Restituisce l'istanza unica di OperatoreSession.
18       * Se l'istanza non esiste, viene creata una nuova istanza.
19       *
20       * @return istanza corrente di OperatoreSession
21       */
22      public static OperatoreSession getInstance() {
23          if (instance == null) {
24              instance = new OperatoreSession();
25          }
26          return instance;
27      }
28
29      /**
30       * Imposta l'operatore registrato per la sessione.
31       *
32       * @param operatore l'operatore registrato da associare alla sessione
33       */
34      public void setOperatore(OperatoreRegistrato operatore) {
35          this.operatore = operatore;
36      }
37
38      /**
39       * Restituisce l'operatore registrato associato alla sessione.
40       *
41       * @return l'operatore registrato della sessione
42       */
43      public OperatoreRegistrato getOperatore() {
44          return this.operatore;
45      }
46
47      /**
48       * Imposta l'ID del centro di monitoraggio per l'operatore registrato.
49       *
50       * @param idcentromonitoraggio l'ID del centro di monitoraggio da
             assegnare
51       */
52      public void setCentroMonitoraggioIdOperatore(int idcentromonitoraggio) {
53          this.operatore.setCentroMonitoraggioId(idcentromonitoraggio);
54      }
55  }
```

- Costruttore Privato: Il costruttore privato (`private OperatoreSession()`) impedisce che nuove

istanze della classe vengano create dall'esterno.

- Metodo `getInstance()`**: Questo metodo verifica se l'istanza della classe è già stata creata: Se `instance` è `null`, crea una nuova istanza. Altrimenti, restituisce l'istanza già esistente.
- Variabile Statica: La variabile `instance` memorizza l'istanza unica della classe.

Il pattern Singleton viene utilizzato , per esempio , per accedere all'operatore registrato corrente e al centro di monitoraggio associato. Ecco un esempio di utilizzo:

```
// Recupera il nome dell'operatore attivo
String operatoreName = OperatoreSession.getInstance().getOperatore().getUserId
    ();

// Recupera il nome del centro di monitoraggio
String nomecentro = stub.ottieniNomeCentro(
    OperatoreSession.getInstance().getOperatore().getCentroMonitoraggioId()
);
```

Vantaggi dell'Implementazione Singleton

- Unicità: Garantisce che ci sia una sola istanza della classe `OperatoreSession`.
- Accesso Globale: Consente l'accesso centralizzato ai dati dell'operatore registrato.
- Consistenza: Tutte le parti dell'applicazione fanno riferimento alla stessa istanza, evitando conflitti di stato.

7.3 Pattern Observer nei JButton

Il Pattern Observer in Java Swing è utilizzato per gestire notifiche tra un oggetto osservato e uno o più osservatori. Nel caso di un JButton, il pulsante agisce come Subject, mentre l'oggetto che implementa ActionListener è l'Observer. Quando il pulsante viene cliccato, il JButton notifica tutti gli osservatori registrati, attivando il metodo actionPerformed dell'interfaccia ActionListener. Questo meccanismo è utile per separare la logica dell'interfaccia utente dalla gestione degli eventi, mantenendo il codice modulare e flessibile.

8 Strutture dati

In questo progetto, vengono utilizzate principalmente strutture dati basate sull'interfaccia `List` di Java, specificamente implementate con la classe `ArrayList`. Questo approccio consente una gestione dinamica e flessibile delle collezioni di oggetti, come stringhe o entità serializzate come Note.

Un esempio significativo dell'uso di `List` è nella classe `DatabaseConnection`, dove viene creata una lista di oggetti `Note` per rappresentare le annotazioni climatiche associate a una specifica area di interesse. Di seguito è riportato il codice per ottenere la lista di note dal database:

```java
/**
 * Ottiene una lista di note associate a un'area di interesse specificata.
 *
 * @param area il nome dell'area di interesse.
 * @return una lista di oggetti Note associati all'area specificata.
 */
public List<Note> getNote(String area) throws DatabaseConnectionException {
    String sql = "SELECT centro_monitoraggio_id, operatore_id,
        data_rilevazione, nota_vento, " +
                 "nota_umidita, nota_pressione, nota_temperatura,
                     nota_precipitazioni, " +
                 "nota_altitudine_ghiacciai, nota_massa_ghiacciai " +
                 "FROM public.parametriclimatici " +
                 "WHERE coordinate_monitoraggio_id = ?";

    List<Note> listaNote = new ArrayList<>();
    int id_area = this.get_id_denominazione_area(area);

    try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
        pstmt.setInt(1, id_area);

        ResultSet rs = pstmt.executeQuery();
        while (rs.next()) {
            Note note = new Note(
                rs.getInt("centro_monitoraggio_id"),
                rs.getInt("operatore_id"),
                rs.getDate("data_rilevazione"),
                rs.getString("nota_vento"),
                rs.getString("nota_umidita"),
                rs.getString("nota_pressione"),
                rs.getString("nota_temperatura"),
                rs.getString("nota_precipitazioni"),
                rs.getString("nota_altitudine_ghiacciai"),
                rs.getString("nota_massa_ghiacciai")
            );
            listaNote.add(note);
        }
    } catch (SQLException e) {
        throw new DatabaseConnectionException("Errore di connessione al
            database: " + e.getMessage());
    }

    return listaNote;
}
```

Nel client, la lista di oggetti `Note` viene recuperata tramite un'architettura RMI, che consente la comunicazione tra il client e il server. La lista viene utilizzata per popolare un'interfaccia grafica con i dati climatici relativi all'area selezionata.

```
/**
 * Costruttore della classe VisualizzaCommentiFrame.
 *
 * @param area L'area per la quale si desidera visualizzare i commenti.
 */
public VisualizzaCommentiFrame(String area) throws DatabaseConnectionException
     {
    this.areaDaCercare = area;
    try {
        Registry registry = LocateRegistry.getRegistry("localhost", 1099);
        stub = (RemoteService) registry.lookup("climatemonitoring.
            RemoteService");

        // Recupera la lista di note dal server
        noteList = stub.getNote(areaDaCercare);

        // Inizializza l'interfaccia grafica con i dati
        initializeUI();
    } catch (RemoteException | NotBoundException e) {
        JOptionPane.showMessageDialog(this,
            "Errore di connessione al server: " + e.getMessage(),
            "Errore", JOptionPane.ERROR_MESSAGE);
        throw new RuntimeException(e);
    }
}
```

- Flessibilità: Le liste consentono di aggiungere, rimuovere e iterare sugli elementi con facilità.
- Serializzazione: Gli oggetti `Note` possono essere serializzati, facilitando il trasferimento tramite RMI.
- Modularità: L'uso di una collezione standard come `List` mantiene il codice leggibile e conforme alle best practice di Java.

9 Costi algoritmici

Per questo progetto, si è scelto di implementare metodi che non abbiano elevati costi computazionali. Ci sono parecchi metodi che hanno costo costante O(1) e altri che effettuano ricerche sul database SERVERLABB tramite Database Connection o richiedono cicli **for** o **while** per esempio per popolare Oggetti Java Swing come **Combobox** o popolare strutture dati come ArrayList (basti guardare il codice della pagina precedente) . Quest'ultimi hanno un costo computazionale paragonabile a O(n).

Di seguito viene mostrato uno dei classici metodi che utilizzano costrutti come cicli **for** , **for-each** per popolare oggetti Java Swing. L'oggetto tutteLeAree è una lista contenente tutte le aree registrate nel database .

```
/**
     * Aggiorna il contenuto del JComboBox con le aree disponibili.
     */

    private void updateComboBox() {
        comboBoxModel.removeAllElements();
        if (tutteLeAree != null && !tutteLeAree.isEmpty()) {
            for (String area : tutteLeAree) {
                comboBoxModel.addElement(area);
            }
        } else {
            comboBoxModel.addElement("Nessuna area disponibile");
        }
    }
```

10 Limiti dell'applicazione

I possibili limiti presenti nell'applicazione potrebbero essere:

- **Sicurezza:** Le informazioni che passano tra client e server non sono crittografate e non vengono effettuati backup dei dati. Potrebbe essere sviluppato un sistema più' efficiente per la protezione di dati sensibili come password.
- **Compatibilità:** Computer più vecchi, con versioni obsolete di Java, potrebbero non supportare il programma correttamente.
- **Connessione:** Se il server non viene avviato e configurato prima dell'avvio del client, quest'ultimo non può funzionare. Inoltre questo sistema è stato progettato con un database locale e non remoto , dunque implica che chi voglia utilizzare questo applicativo necessiti di configurare manualmente il database , il che è risulta non intuitivo per chi non ha conoscenze del mondo informatico . In generale il processo di configurazione non è user-friendly , a differenza dell'interfaccia.
- **Funzionalità** Per quanto riguarda il rapporto di lavoro tra un operatore-centro manca una funzionalità che preveda lo scenario in cui l'utente Mario Rossi concluda il suo contratto lavorativo con il Centro di Monitoraggio laziale per iniziare un nuovo contratto lavorativo con il Centro di Monitoraggio Campano (già pre-esistente nel database) . Invece un operatore può' concludere il suo rapporto lavorativo con un centro solo ed esclusivamente se ne crea uno nuovo , scenario meno verosimile di quello non implementato .

11 Strumenti utilizzati

Il progetto Climate Monitoring è stato sviluppato secondo questa versione di Java:

java version "19" 2022-09-20
Java(TM) SE Runtime Environment (build 19+36-2238)

Il database PostgreSQL è stato sviluppato con questa versione:

PostgreSQL 16.4, compiled by Visual C++ build 1940, 64-bit

Il database è configurato sulla porta 5432, con hostname: `localhost`. Il codice è stato sviluppato con IntelliJ IDEA. La documentazione è stata prodotta con:

https://it.overleaf.com/LaTeX tramite editor Overleaf

https://plantuml.com/PlantUML per generare i diagrammi

12 Bibliografia

How to write a technical manual : https://www.proprofskb.com/blog/write-technical-manual/