Laboratorio B

Insubria

CLIMATE MONITOR

MANUALE TECNICO

Università degli studi dell'Insubria - Laurea Triennale Informatica Progetto Laboratorio B: Climate Monitor

SOMMARIO

INTRODUZIONE

Climate Monitor è un software di monitoraggio delle temperature globali. È un sistema di database che raccoglie e analizza dati da fonti affidabili, come le stazioni meteorologiche, i satelliti e altre fonti di dati climatici. Il software è stato sviluppato attraverso **Java 17** implementando l'interfaccia grafica di **NetBeans** e testato su **Windows 10/ Windows 11** e **MacOS** (per quest'ultimo sono presenti delle limitazione che verranno spiegate successivamente)**.**

AMBIENTE UTILIZZATO

1. NetBeans

Per la realizzazione del software e delle relative interfacce grafiche è stato utilizzato **NetBeans**, un ambiente di sviluppo open source che fornisce strumenti e funzionalità per la creazione di applicazioni software. Una delle caratteristiche principali di NetBeans è la sua interfaccia utente intuitiva e personalizzabile, che semplifica la scrittura, il debug e il testing del codice.

2. PostgreSQL

PostgreSQL è un sistema di gestione di database relazionale a oggetti pen source. Offre funzionalità avanzate come il supporto per tipi di dati complessi, estensioni personalizzate e transazioni ACID. È utilizzato in molte applicazioni web, analisi dei dati e altre soluzioni enterprise grazie alla sua affidabilità e performance.

3. pgAdmin

pgAdmin è uno strumento di gestione open source per PostgreSQL. Fornisce l'interfaccia grafica per amministrare database, eseguire query SQL, gestire oggetti del database e monitorare le prestazioni.

LIBRERIE UTILIZZATE

Per lo sviluppo del software sono state utilizzate le seguenti librerie:

1. Java Swing

Si tratta di una libreria di componenti grafici per la creazione di interfacce grafiche utente (GUI) in applicazioni Java. Offre una vasta gamma di componenti grafici avanzati, tra cui bottoni, etichette, caselle di testo, menu, tabelle, alberi e molti altri. I componenti

grafici Swing possono essere personalizzati attraverso la modifica dei loro aspetti visivi e delle loro proprietà di comportamento, dimensioni, font eccetera tramite il pannello properties, consentendo agli sviluppatori di creare interfacce grafiche personalizzate e di alta qualità.

2. Java.io

E' un pacchetto in Java che fornisce classi e metodi per gestire operazioni di input e output.

3. Java.util

E' un pacchetto in Java che fornisce una vasta gamma di classi e interfacce di utilità per la gestione di collezioni, date, tempi, eventi, randomizzazione e altre operazioni comuni come StringTokenizer, Timer, Arrays, Arraylist e molte altre. Queste classi sono progettate per semplificare il lavoro comune in Java e offrire funzionalità aggiuntive per una varietà di operazioni.

4. Java.sql

La libreria java.sql in Java fornisce le classi e le interfacce necessarie per connettersi a database relazionali, eseguire query SQL e gestire i risultati. Essa include strumenti per gestire connessioni, dichiarazioni SQL, set di risultati e gestire transazioni, facilitando l'interazione con vari sistemi di gestione di database (DBMS).

5. PostgreSQL 42.7.jar

E' una libreria utilizzata per il collegamento al database PostgreSQL. Si tratta infatti di un **driver JDBC** che permette alle applicazioni interagire con il database utilizzato.

COMPONENTI GRAFICI

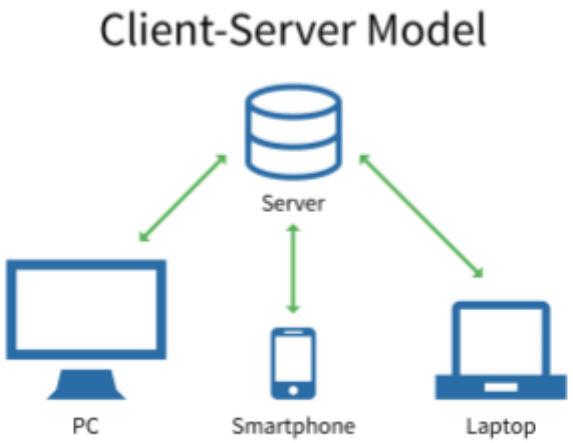
Di seguito verranno riportati i componenti grafici utilizzati per la realizzazione delle interfacce tramite la funzione offerta da Netbeans di **drag and drop**:

1. **Panel** → Pannelli all'interno dell' interfaccia che permettono di aggiungere altri componenti grafici;
2. **Label** → Aree di testo non editabili;
3. **Button** → Si tratta di uno dei componenti grafici più comuni e sono utilizzati per fornire un'interfaccia utente interattiva;
4. **Check box** → E' un componente grafico che consente all'utente di selezionare una o più opzioni da un insieme di opzioni disponibili;
5. **Radio button** → Consente all'utente di selezionare una sola opzione da un insieme di opzioni disponibili;
6. **Button group** → E' un insieme di pulsanti di opzione che sono associati insieme in modo che solo uno di essi possa essere selezionato alla volta;

7. **Combo Box** → E' un componente grafico che consente all'utente di selezionare un'opzione da un elenco a discesa.
8. **Text field** → Consente all'utente di inserire del testo editabile;
9. **Table** → E' un componente grafico che consente di visualizzare dati in formato tabellare.

STRUTTURA DEL PROGRAMMA

CLIENT-SERVER



Per l'implementazione del progetto è stato utilizzato un approccio **Client-Server**.
Quest'ultimo è un modello di rete in cui i **Client** (utenti o dispositivi) si connettono a un **Server centrale** per accedere a risorse, servizi o dati.
Il client invia richieste al server, che le elabora e restituisce le risposte appropriate.
Questa struttura permette di **centralizzare la gestione delle risorse**, migliorare la sicurezza e facilitare la manutenzione. I server possono *gestire molteplici client contemporaneamente*, bilanciando il carico di lavoro. Questo modello è comunemente utilizzato in applicazioni web, email, file sharing e database, dove un server potente serve numerosi client leggeri, ottimizzando l'efficienza e la scalabilità del sistema.

RMI

RMI (acronimo di **Remote Method Invocation**) è una tecnologia di Java che permette la comunicazione tra oggetti in esecuzione su macchine diverse in una rete. RMI consente a un programma Java di invocare metodi su un oggetto remoto come se fosse locale. Questo è particolarmente utile per creare applicazioni distribuite e per la comunicazione tra diversi componenti di un sistema che possono trovarsi su server distinti.

Componenti di RMI

1. **Stubs e Skeletons**: RMI utilizza stubs (nel client) e skeletons (nel server) per comunicare. Il client chiama un metodo sullo stub, che a sua volta invia la richiesta al server. Il skeleton riceve la richiesta, la processa e invia la risposta al client tramite il stub.

2. **Registry**: Un registro RMI è un servizio di directory che consente ai Client di trovare gli oggetti remoti. Gli oggetti remoti devono essere registrati con un nome univoco nel registro, permettendo ai Client di cercare e connettersi a questi oggetti.
3. **Serializzazione degli oggetti**: Gli argomenti e i valori di ritorno dei metodi invocati devono essere serializzati, cioè convertiti in un formato che può essere trasmesso attraverso la rete. La serializzazione permette di trasferire oggetti complessi, compresi quelli che contengono riferimenti ad altri oggetti.

Processo di RMI

1. **Creazione dell'oggetto remoto**: Il server crea un'istanza dell'oggetto remoto che implementa un'interfaccia remota.
2. **Esportazione dell'oggetto remoto**: L'oggetto remoto viene esportato per renderlo disponibile ai Client.
3. **Registrazione dell'oggetto**: L'oggetto remoto viene registrato con il servizio di registro RMI usando un nome univoco.
4. **Client lookup**: Il Client cerca l'oggetto remoto nel registro usando il nome univoco.
5. **Invocazione del metodo remoto**: Il Client invoca metodi dell'oggetto remoto tramite lo stub, che gestisce la comunicazione con il server.

DATABASE

Come precedentemente indicato, per la progettazione del database è stato impiegato **PostgreSQL**, utilizzando l'interfaccia grafica di **pgAdmin 4.** Una volta avviato il server eseguibile, l'applicazione opererà automaticamente sul computer locale, poiché il protocollo *Client-Server* viene simulato sulla stessa macchina.

La schermata a destra mostra la pagina grafica di inizializzazione del server, dove sono specificati i campi di connessione.
L'unico parametro modificabile è la password.

COMPILAZIONE DEL DATABASE (Windows)

Una volta estratto il file .zip del programma, per il corretto funzionamento del database e il suo riempimento di dati relativi alle rilevazioni e alle tabelle SQL che lo compongono, è necessario inserire la cartella *Temp* all'interno del *Disco C:* del nostro computer.
Una volta eseguita questa procedura sarà sufficiente avviare correttamente il nostro programma (prima server e poi client) per creare e riempire automaticamente le tabelle e avendo così il programma pronto per l'utilizzo.

Sistema operativo macOS o errori di compilazione

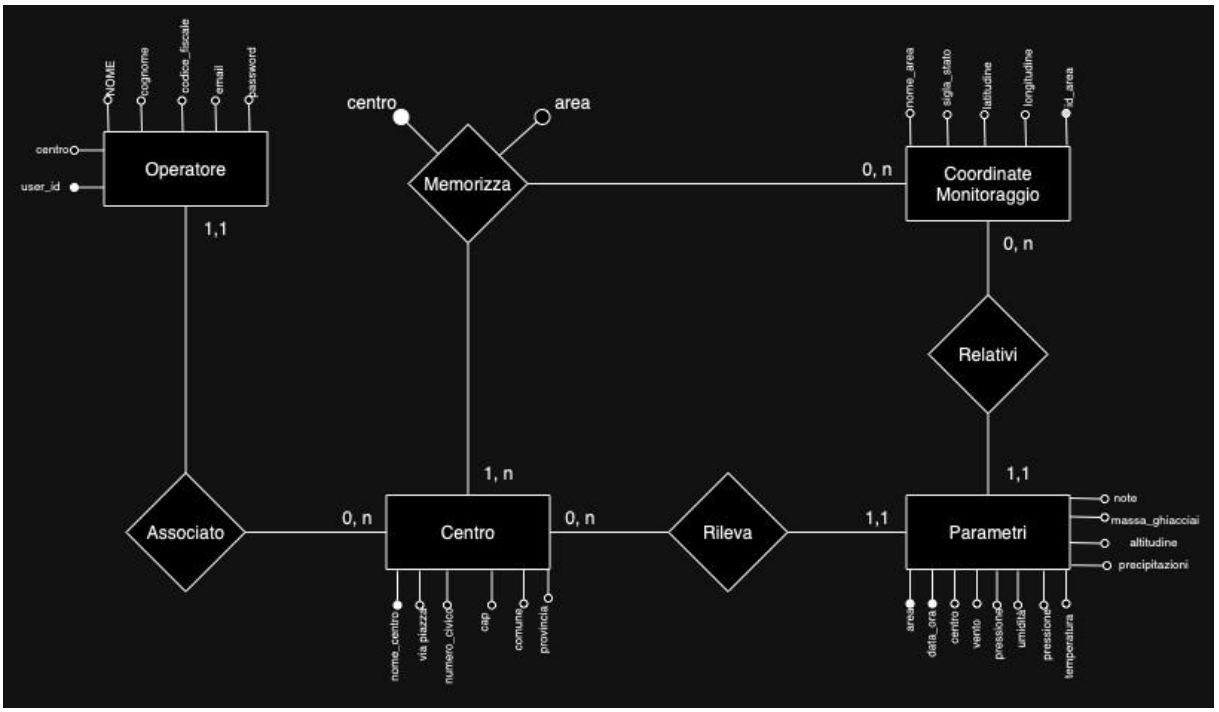
Nei sistemi macOS il percorso file che cerca la cartella *Temp* è diverso da Windows. Per questo, durante la compilazione, può capitare frequentemente di riscontrare dei problemi.

1. **Fase 1:** Aprire pgAdmin e seguire il percorso *Server* > *localhost* e inserire la password;
2. **Fase 2:** Aprire il percorso *Database* > *db_climatemonitoring* > *Schemas* > *public* > *Tables;*
3. **Fase 3:** Tasto destro sulla tabella da compilare (attenzione bisogna eseguire questa procedura su tutte le tabelle), selezionare *Import/Export Data*;
4. **Fase 4:** Selezionare *Import*, *In File Name* selezionare il percorso del file .txt relativo alla tabella che si sta compilando. Come format selezionare *text* e come encoding selezionare *utf 8*;
5. **Fase 5:** Una volta dato l'ok, se il programma risponde correttamente, procedere con le tabelle mancanti fino a completare la compilazione.

NB. Ricordarsi di importare le tabelle nell'ordine corretto per non riscontrare problemi di integrità e vincoli. L'ordine corretto è il seguente: AreaInteresse , Operatore, CentroMonitoraggio, Parametri, Memorizza.

Modello ER

Il modello seguente rappresenta lo schema ER relativo al database



SCHEMA LOGICO

centriMonitoraggio (nome_centro, via_ piazza, numero_civico, cap, comune, provincia)

coordinatemonitoraggio (geocode, nome_area, sigla_stato, latitudine, longitudine)

memorizza (centro^{nome_centro}, area^{geocode})

operatoriregistrati (nome, cognome, codice_fiscale, email, user_id, password, centro^{nome_centro})

parametriclimatici (centro^{nome_centro}, area^{geocode}, data_ora, vento, note_vento, umidità, note_umidità, pressione, note_pressione, temperatura, note temperatura, precipitazioni, note_precipitazioni, altitudine_ghiacciai, note_altitudine, massa_ghiacciai, note_massa)

Ulteriori diagrammi UML sono presenti all'interno della cartella zip "Diagrammi" del programma per la spiegazione delle funzioni principali del programma.

PACKAGES

Il progetto è suddiviso in **packages** (nello specifico in **5 pacchetti**) ovvero raggruppamenti di classi, interfacce ecc. utilizzati per organizzare il codice in maniera modulare e strutturata.
I package aiutano a evitare conflitti di nomi, migliorano la manutenibilità del codice e controllano l'accesso alle classi e ai metodi.
Di seguito è riportata la lista dei package e delle classi contenute in essi:

1. **Package Server**
    a. *ConnessioneDatabase.java;*
    b. *CodiceSQL;*
2. **Package Gestione_Classe**
    a. *GestioneArea;*
    b. *GestioneCentro;*
    c. *GestioneOperatore;*
    d. *GestioneRilevazione;*
3. **Package ClimateMonitoring**
    a. *ClimateMonitor;*
    b. *AreaInteresse;*
    c. *Operatore;*
    d. *CentroMonitoraggio;*
    e. *Rilevazione;*
4. **Package Grafica**
    a. *AccediCittadino;*
    b. *AccediOperatore;*
    c. *DatiArea;*
    d. *InserimentoRilevazione;*
    e. *RegistraCentro;*
    f. *RegistraOperatore;*
    g. *SceltaModalita;*
5. **Package Interfacce**
    a. *InterfacciaArea;*
    b. *InterfacciaCentro;*
    c. *InterfacciaOperatore;*
    d. *InterfacciaRilevazione.*

ITERAZIONI TRA LE CLASSI GRAFICHE

Le classi grafiche comunicano tra di loro nel seguente modo:

- **InterfacciaArea** con **AccediCittadino** e **DatiArea**;
- **InterfacciaCentro** con **RegistraCentro**;

- **InterfacciaOperatore** con **AccediOperatore** e **RegistraOperatore**;
- **IterfacciaRivelazione** con **DatiArea** e **InserimentoRilevazione**.

LATO SERVER

PACKAGE *SERVER*

1. ConnessioneDatabase.java

Questa classe si occupa della connessione al database. Oltre al costruttore vuoto per effettuare la connessione troviamo i seguenti metodi:

1. inizializza_dati()

Metodo utilizzato per inizializzare i dati con cui si effettuerà la connessione al database

2. Connection getConnection() throws SQLException[1]

Metodo che ritorna un oggetto di tipo connessione (Connection) utilizzando il pattern singleton ovvero design pattern creazionale che assicura che una classe abbia una sola istanza e fornisca un punto di accesso globale a questa istanza. Questo significa che, indipendentemente da quante volte venga richiesto un oggetto di quella classe, verrà sempre restituita la stessa istanza.

3. modificaConnessione()

Metodo che chiude la connessione al database e lo statement, creando un altro oggetto di tipo (Connection) per una nuova connessione al database e altri due statement nuovi per utilizzare la nuova connessione che viene ritornata.

4. getStatement()

Metodo che ritorna un oggetto di tipo Statement che serve per effettuare le interrogazioni/query al database.

5. setURL()

Metodo Che serve per creare un nuovo URL per effettuare la connessione al database aggiungendone il nome del DB.

6. esiste_db()

Metodo che controlla se il database esiste già su postgres (vuol dire che è già stato creato in precedenza) ritornando un booleano.

7. crea_db()

Metodo che serve per creare il database su postgres tramite l'apposita query.

8. crea_tabelle()

Metodo che serve alla creazione delle tabelle da inserire nel database su postgres tramite le apposite query.

[1] *SQLException lancia le eccezioni che riguardano operazioni errate sul database*

2. CodiceSQL

Questa classe incorpora tutte le query eseguite sul database, inclusi i comandi per la creazione del database e delle tabelle, l'inserimento dei dati e la loro interrogazione. In particolare, essa comprende le seguenti query:

1. *public static final String creazione_db*

Questa query SQL **crea** il database.

2. *public static final String query_creazione*

Questa query SQL crea **cinque tabelle** nel database: *CoordinateMonitoraggio*, *CentriMonitoraggio, Memorizza, OperatoriRegistrati, e ParametriClimatici*, definendo le rispettive colonne, chiavi primarie e vincoli di integrità referenziale.

3. *public static final String select_query[]*

Questo **array** di stringhe contiene query SQL per eseguire **operazioni di selezione** sul database, come autenticare utenti, ottenere sigle di stati, trovare geocodici e dettagli delle aree in base a vari criteri.

4. *public static final String insert_query[]*

Questo array di stringhe contiene query SQL per **inserire** nuovi record nelle tabelle *OperatoriRegistrati, CentriMonitoraggio, Memorizza, e ParametriClimatici* del database.

5. *public static final String parameters_query[]*

Questo array di stringhe contiene query SQL per eseguire operazioni sui parametri climatici, come ottenere il valore **massimo**, **minimo**, **medio** e il **valore più frequente** di un parametro specifico, oltre a **selezionare** dati climatici per un'area specifica.

6. *public static final String update_query[]*

Questa query SQL **aggiorna** il campo centro di un record nella tabella *OperatoriRegistrati* per un determinato *user_id*.

7. *public static final String control_query_operatori[]*

Questo array di stringhe contiene query SQL per selezionare tutti gli *user_id*, le *email* e i *codici fiscali* dalla tabella *OperatoriRegistrati* e tutti i nomi dei centri dalla tabella *CentriMonitoraggio*.

8. *public static final String control_query_centri[]*

Questo array di stringhe contiene query SQL per selezionare tutti i nomi dei centri dalla tabella *CentriMonitoraggio* e per ottenere i *dettagli di indirizzo* (*comune, provincia, via/piazza, numero civico, CAP)* dei centri.

9. *public static final String control_query_area[]*

Questo array di stringhe contiene query SQL per selezionare le sigle degli stati dalla tabella *CoordinateMonitoraggio*, per ottenere i nomi delle aree con le rispettive sigle di stato dalla stessa tabella, e per recuperare gli identificatori delle aree memorizzate dalla tabella *Memorizza*.

PACKAGE *INTERFACCE*

All'interno di questo package troviamo le interfacce che fungono da stub per il Client per accedere all'oggetto remoto tramite RMI, dunque consentendo al Client di avere un riferimento all'interfaccia per l'accesso all'oggetto remoto.

Eccezioni sollevate

Tutti i metodi delle interfacce lanciano l'eccezione **SQLException** che viene sollevata quando si verifica un errore durante l'accesso o l'elaborazione dei dati in un database utilizzando JDBC. Questo può accadere ad esempio quando la sintassi della query è errata, si tenta di accedere a una tabella inesistente, o ci sono problemi di connessione al database e ***RemoteException*** che viene sollevata quando si verificano errori durante l'invocazione di metodi remoti in Java tramite RMI. Questo può accadere a causa di problemi di connettività di rete, errori nel server remoto o errori nel trasporto dei dati.

All'interno di questo package sono presenti le seguenti classi.

1. InterfacciaArea.java

Di seguito è riportata la lista dei metodi:

1. *public void inserisci_area(String centro, String lista_aree);*
2. *public String ritorna_errore(int numero_aree);*
3. *public boolean area_gia_inserita(int geocode);*
4. *public String ritorna_errore_lat_long();*
5. *public String ricerca_coordinate_avanzata();*
6. *public String ricerca_coordinate();*
7. *public String ricerca_nome(String str, String stato);*
8. *public String lista_stati();*
9. *public int recuperaGeocode(String sigla, String nome);*
10. *public void setArea(AreaInteresse a);*

2. InterfacciaCentro.java

Di seguito è riportata la lista dei metodi:

1. *public void registra_centroMonitoraggio()*
2. *public String ritorna_errore()*
3. *public void aggiungi_centro_a_operatore(String nome_centro, int id)*
4. *public String aree_centro(String nome_centro)*
5. *public void setCentro(CentroMonitoraggio c)*

3. InterfacciaOperatore.java

Di seguito è riportata la lista dei metodi:

1. *public void registra_operatore()*
2. *public Operatore accesso(int userid, String password)*
3. *public String ritorna_errore(boolean flag)*
4. *public void setOperatore(Operatore operatore)*

4. InterfacciaRilevazione.java

Di seguito è riportata la lista dei metodi:

1. *public void registra_rilevazione()*
2. *public String dati_tabella(int geocode, int i)*
3. *public int[] trovaMassimo(int geocode)*
4. *public int[] trovaMinimo(int geocode)*
5. *public double[] trovaMedie(int geocode)*
6. *public int[] trovaMode(int geocode)*
7. *public String controlli_note(String[] note)*
8. *public void setRilevazione(Rilevazione r)*

PACKAGE *GESTIONE_CLASSI*

Questo package contiene le classi che **implementano** tutti i metodi dell'interfaccia.
Come già citato in precedenza, anche queste classi sollevano le eccezioni **SQLException** ***RemoteException***.

All'interno di questo package sono presenti le seguenti classi.

1. GestioneArea.java

Questa classe gestisce ed effettua le operazioni sull'oggetto di tipo *Area*.java.
Di seguito sono riportati i metodi che implementano l'interfaccia **InterfacciaArea.java**.

1. ***public synchronized void inserisci_area(String centro, String lista_aree)***

Questo metodo inserisce aree associate a un centro specifico nel database. Estrae il nome e la sigla delle aree dalla stringa fornita, esegue una query per ottenere il geocode corrispondente nel database, quindi inserisce l'associazione tra il centro e l'area utilizzando il geocode.

2. ***public synchronized String ritorna_errore(int numero_aree)***

Questo metodo controlla se ci sono errori nei dati inseriti per un'area. Verifica se il numero di aree è superiore a 1, se la sigla dello stato è di 2 lettere, se lo stato è presente nel database, se il nome dell'area è sufficientemente lungo e se l'area è presente nel database. Restituisce un messaggio di errore appropriato se uno di questi controlli fallisce, altrimenti restituisce una stringa vuota.

3. ***public synchronized boolean area_gia_inserita(int geocode)***

Questo metodo controlla se un'area è già stata inserita nel database confrontando il geocode fornito con gli geocode presenti nella tabella delle aree. Restituisce true se l'area è già stata inserita, altrimenti false.

4. ***private synchronized boolean area_trovata(String paese, String sigla)***

Questo metodo verifica se un'area è presente nel database confrontando il nome dell'area e la sigla dello stato forniti con quelli presenti nella tabella delle aree. Restituisce true se l'area è trovata, altrimenti false.

5. ***private synchronized boolean stato_presente(String sigla)***

Questo metodo verifica se una sigla di stato è presente nel database confrontando la sigla fornita con quelle presenti nella tabella delle aree. Restituisce true se la sigla di stato è presente, altrimenti false.

6. ***public synchronized String ritorna_errore_lat_long()***

Questo metodo controlla se le coordinate geografiche inserite (latitudine e longitudine) sono valide. Restituisce un messaggio di errore se le coordinate non sono corrette (fuori dai limiti consentiti), altrimenti restituisce una stringa vuota.

7. ***public synchronized String ricerca_coordinate_avanzata()***

Questo metodo ricerca aree geografiche nel database che si trovano entro un raggio di 0.15 gradi dalla latitudine e dalla longitudine dell'area fornita. Restituisce una stringa contenente il nome dell'area, la sigla dello stato, la latitudine e la longitudine di ciascuna area trovata, separati da "%" e terminati da "!".

8. ***public synchronized String ricerca_coordinate()***

Questo metodo ricerca nel database le aree geografiche con le coordinate esatte corrispondenti a quelle fornite. Restituisce una stringa contenente il nome dell'area, la sigla dello stato, la latitudine e la longitudine di ciascuna area trovata, separati da "%" e terminati da "!".

9. ***public synchronized String ricerca_nome(String str, String stato)***

Questo metodo ricerca nel database le aree geografiche con un nome che corrisponde alla stringa fornita e alla sigla dello stato specificato. Restituisce una stringa contenente il nome dell'area, la sigla dello stato, la latitudine e la longitudine di ciascuna area trovata, separati da "%" e terminati da "!".

10. ***public String lista_stati()***

Questo metodo ottiene una lista di sigle degli stati presenti nel database e le restituisce come una stringa separata da "-".

11. ***public synchronized int recuperaGeocode(String sigla, String nome)***

Questo metodo recupera il geocode di un'area nel database, dato il nome e la sigla dello stato. Esegue una query per cercare l'area con la sigla e il nome forniti e restituisce il geocode corrispondente.

12. ***public void setArea(AreaInteresse a)***

Questo metodo imposta l'oggetto '*area*' con l'oggetto '*a*' fornito come parametro.

2. GestioneCentro.java

Questa classe gestisce ed effettua le operazioni sull'oggetto di tipo *Centro.java*.
Di seguito sono riportati i metodi che implementano l'interfaccia **InterfacciaCentro.java**.

1. ***public synchronized void registra_centroMonitoraggio()***

Questo metodo registra un nuovo centro di monitoraggio nel database utilizzando i dati forniti dall'oggetto '*ct*'. Esegue una query di inserimento nella tabella *CentriMonitoraggio* utilizzando i valori dei campi del centro di monitoraggio.

2. ***public synchronized String ritorna_errore()***

Questo metodo controlla se ci sono errori nei dati del centro di monitoraggio. Verifica la lunghezza e la correttezza dei valori del nome, della via/piazza, del numero civico, del CAP, del comune e della provincia. Controlla anche se il nome del centro e l'indirizzo sono già presenti nel database. Restituisce un messaggio di errore appropriato se uno di questi controlli fallisce, altrimenti restituisce una stringa vuota.

3. ***private synchronized boolean centro_gia_presente(String nome)***

Questo metodo controlla se un centro di monitoraggio con il nome fornito è già presente nel database. Esegue una query per cercare il nome del centro nella tabella dei centri di monitoraggio e restituisce true se il centro è già presente, altrimenti false.

4. ***private synchronized boolean indirizzo_gia_presente(int numero, String cap, String comune, String provincia, String via)***

Questo metodo controlla se l'indirizzo è già associato a un centro di monitoraggio nel database. Esegue una query per cercare l'indirizzo (numero civico, CAP, comune, provincia e via/piazza) nella tabella dei centri di monitoraggio e restituisce true se l'indirizzo è già presente, altrimenti false.

5. ***public synchronized void aggiungi_centro_a_operatore(String nome_centro, int id)***

Questo metodo aggiorna il campo "centro" per un operatore registrato nel database, associandolo al centro di monitoraggio specificato dal nome. Esegue una query di aggiornamento nella tabella degli operatori registrati utilizzando il nome del centro e l'ID dell'operatore come parametri.

6. ***public synchronized String aree_centro(String nome_centro)***

Questo metodo recupera le aree associate a un centro di monitoraggio specificato dal nome. Esegue una query nel database per ottenere le aree associate al centro specificato e restituisce una stringa contenente la sigla dello stato e il nome di ciascuna area separati da "-", e le aree separate da ",".

7. ***public synchronized void setCentro(CentroMonitoraggio c)***

Questo metodo imposta l'oggetto '*ct*' con l'oggetto '*c*' fornito come parametro, che rappresenta un centro di monitoraggio.

3. GestioneOperatore.java

Questa classe gestisce ed effettua le operazioni sull'oggetto di tipo *Operatore.java*.
Di seguito sono riportati i metodi che implementano l'interfaccia **InterfacciaOperatore.java**.

1. ***public synchronized void registra_operatore()***

Questo metodo registra un nuovo operatore nel database utilizzando i dati forniti dall'oggetto '*op*'. Esegue una query di inserimento nella tabella degli operatori registrati utilizzando i valori dei campi dell'operatore.

2. ***public synchronized Operatore accesso(int userid, String password)***

Questo metodo gestisce l'accesso di un operatore nel sistema. Esegue una query nel database per cercare un operatore con l'ID utente e la password forniti. Se trova una corrispondenza, crea un oggetto *Operatore* con i dati recuperati e lo restituisce. Altrimenti, restituisce null.

3. ***public synchronized String ritorna_errore(boolean flag)***

Questo metodo verifica la validità dei dati inseriti durante la registrazione di un operatore. Controlla la lunghezza e il formato del nome, cognome, codice fiscale, email, userid e password. Restituisce un messaggio di errore appropriato se i dati non sono validi. Controlla anche se il codice fiscale, l'email e l'userID sono già presenti nel database e se il nome del centro è valido. Se il flag è true, verifica anche se il nome del centro è presente nel database. Restituisce una stringa vuota se non ci sono errori.

4. ***private synchronized boolean userid_gia_presente(int id)***

Questo metodo controlla se un determinato ID utente è già presente nel database degli operatori registrati. Esegue una query nel database per cercare l'ID utente specificato e restituisce true se lo trova, altrimenti restituisce false.

5. ***private synchronized boolean email_gia_presente(String email)***

Questo metodo verifica se un'email è già presente nel database degli operatori registrati. Esegue una query nel database per cercare l'email specificata e restituisce true se la trova, altrimenti restituisce false.

6. ***private synchronized boolean validita_cod_fiscale(String cf)***

Questo metodo verifica la validità di un codice fiscale controllando se contiene esattamente 9 lettere e 7 cifre. Analizza ogni carattere del codice fiscale e conta il numero di lettere e cifre presenti. Restituisce true se il codice fiscale è valido secondo questi criteri, altrimenti restituisce false.

7. ***private synchronized boolean centro_presente(String nome)***

Questo metodo controlla se un centro è già presente nel database dei centri di monitoraggio. Esegue una query nel database per cercare il nome del centro specificato e restituisce true se lo trova, altrimenti restituisce false.

8. ***private synchronized boolean codice_fiscale_gia_presente(String cf)***

Questo metodo verifica se un dato codice fiscale è già presente nel database degli operatori registrati. Esegue una query nel database per cercare il codice fiscale specificato e restituisce true se lo trova, altrimenti restituisce false.

9. ***public void setOperatore(Operatore operatore)***

Questo metodo imposta un'istanza della classe "Operatore" all'interno dell'oggetto corrente.

4. GestioneRilevazione.java

Questa classe gestisce ed effettua le operazioni sull'oggetto di tipo *Rilevazione.java*.
Di seguito sono riportati i metodi che implementano l'interfaccia **InterfacciaRilevazione.java**.

1. ***public synchronized void registra_rilevazione()***

Questo metodo registra una rilevazione meteorologica nel database utilizzando un'istruzione SQL predefinita. I dati della rilevazione sono estratti dall'oggetto "ril" e inseriti nelle colonne corrispondenti della tabella del database.

2. ***public synchronized String dati_tabella(int geocode, int i)***

Questo metodo recupera i dati dalla tabella del database utilizzando un'istruzione SQL predefinita con parametri specificati. I risultati sono concatenati in una stringa separata da delimitatori e restituiti.

3. ***public synchronized int[] trovaMassimo(int geocode)***

Questo metodo cerca il massimo valore di una serie di parametri specifici per un dato geocode nel database. Utilizza un'istruzione SQL predefinita e restituisce un array di interi contenenti i valori massimi trovati.

4. ***public synchronized int[] trovaMinimo(int geocode)***

Questo metodo cerca il massimo valore di una serie di parametri specifici per un dato geocode nel database. Utilizza un'istruzione SQL predefinita e restituisce un array di interi contenenti i valori minimi trovati.

5. ***public synchronized double[] trovaMedie(int geocode)***

Questo metodo cerca il massimo valore di una serie di parametri specifici per un dato geocode nel database. Utilizza un'istruzione SQL predefinita e restituisce un array di interi contenenti i valori medi trovati.

6. ***public synchronized int[] trovaMode(int geocode)***

Questo metodo cerca il massimo valore di una serie di parametri specifici per un dato geocode nel database. Utilizza un'istruzione SQL predefinita e restituisce un array di interi contenenti i valori delle mode trovate.

7. ***public synchronized String controlli_note(String[] note)***

Questo metodo controlla se la lunghezza delle note fornite supera i 256 caratteri per ciascun parametro. Restituisce il parametro corrispondente al primo superamento o una stringa vuota se nessun parametro supera il limite.

8. ***public synchronized void setRilevazione(Rilevazione r)***

Questo metodo imposta un'istanza della classe *Rilevazione* all'interno dell'oggetto corrente.

PACKAGE *CLIMATEMONITORING*

1. ClimateMonitor.Java

La classe **ClimateMonitor** si occupa di lanciare il *Main*.
Questa pagina si occupa della generazione dei file .jar. All'interno di questa classe viene scelta quale delle due modalità eseguire (client o server).
All'interno della classe infatti può essere richiamato il metodo **Scelta_modalita.main(args)** oppure **AvvioServer.main(args).**

2. AreaInteresse.Java

La classe **AreaInteresse** si occupa di raccogliere le aree di interesse.
Tra i parametri troviamo nome, id_stato, latitudine, longitudine, id_areaInteresse.

3. Operatore.Java

La classe **Operatore** si occupa della registrazione dell'utente.
Tra i parametri troviamo nome, cognome, codice fiscale, email, password, centro di monitoraggio, user ID.

4. CentroMonitoraggio.Java

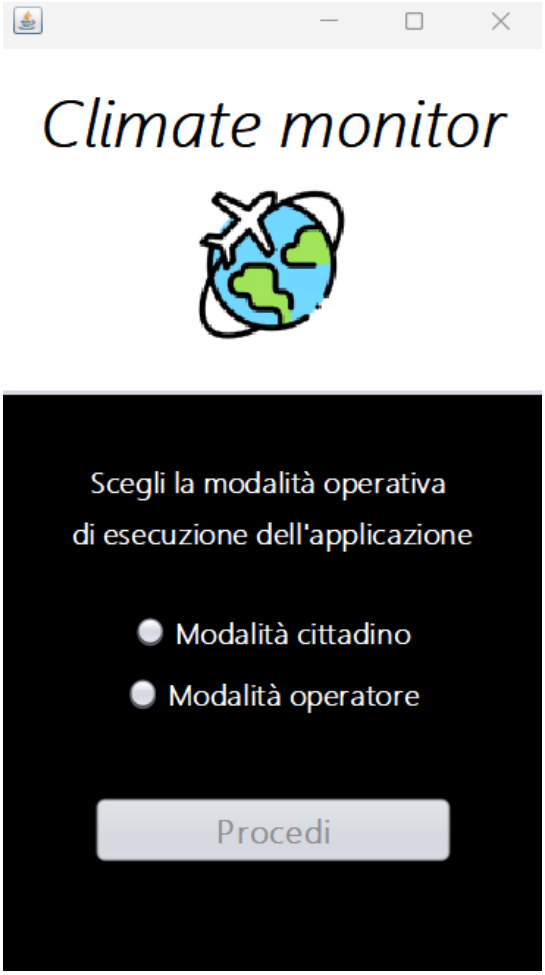
La classe **CentroMonitoraggio** si occupa dell'inserimento di un nuovo centro di monitoraggio.
Tra i parametri troviamo nome centro, via / piazza, comune, provincia, aree, numero civico, cap.

5. Rilevazione.Java

La classe **Rilevazione** si occupa dell'inserimento delle rilevazioni di una determinata area di interesse.
Tra i parametri troviamo String[] note, centro, area, data, vento, umidità, pressione, temperatura, precipitazioni, altitudine ghiacciai, massa ghiacciai.

LATO CLIENT

1. INIZIO



Struttura grafica

Per la realizzazione della pagina di login dell' operatore sono stati usati due pannelli: il primo dedicato al nome del software e il secondo contenente:

1. Le label per le varie intestazioni;
2. I textfield per l'inserimento dell'ID e della password;
3. Il bottone per accedere;
4. In caso è necessaria la registrazione è possibili cliccare sulla label "Registrati" che inoltra l'utente sulla schermata di registrazione;
5. "Back" per tornare alla pagina precedente

Codice

La pagina viene creata tramite il costruttore e viene inizializzata graficamente tramite il metodo **initComponent()**, questo vale per tutte le pagine grafiche spiegate di seguito. Nella classe grafica **SceltaModalita.java** non troviamo veri e propri metodi ma al momento della scelta della modalità viene creato un oggetto di tipo **AccediOperatore** se si sceglie la modalità operatore, altrimenti un oggetto **AccediCittadino**.

2. LOGIN

Struttura grafica

Per la realizzazione della pagina di login dell' operatore sono stati usati due pannelli: il primo dedicato al nome del software e il secondo contenente:

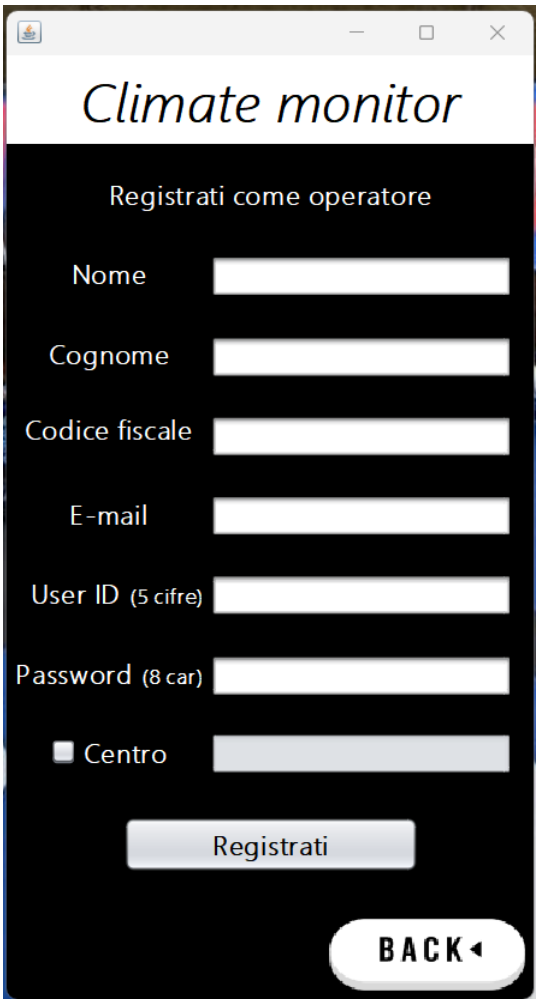
1. Le label per le varie intestazioni;
2. I textfield per l'inserimento dell'ID e della password;
3. Il bottone per accedere;
4. In caso è necessaria la registrazione è possibili cliccare sulla label "Registrati" che inoltra l'utente sulla schermata di registrazione;
5. "Back" per tornare alla pagina precedente

Codice

Nella classe grafica **AccediOperatore.java** troviamo i metodi di accesso che leggono i contenuti nel database controllando se i valori inseriti sono validi (nel caso di errore verranno mostrati pop-up di errore), dopodiché crea un oggetto di tipo **operatore** tramite il costruttore di due parametri (user ID e password).

Tramite il metodo accedi si verifica se l'operatore è esistente e si procede alla creazione del centro (se non è presente) oppure all'inserimento dei dati della rilevazione.

3. REGISTRAZIONE



Struttura grafica

La pagina di registrazione è stata realizzata utilizzando due pannelli:

1. Il primo dedicato al nome del programma e il logo ad esso dedicato;
2. Il secondo contenente tutte le label di intestazione e text fields in cui inserire i dati dell'operatore che intende registrarsi.

Se l'operatore dispone già di un centro può spuntare la checkbox che attiva automaticamente il textfield dedicato per scrivere il centro in cui lavora.
Tutti i dati relativi agli operatori registrati vengono salvati in un file di testo che viene utilizzato nella fase ci login per far accedere ciascuno di essi.

Codice

Nella classe grafica **RegistraOperatore.java** è stato creato il metodo controlli che richiama il metodo **resoconto_controlli** della classe **Operatore.java**.
In base al numero restituito da quest'ultimo, tramite il metodo controlli verrà comunicato all'operatore quale errore si è verificato sui parametri inseriti.
In questa classe grafica troviamo anche il metodo registrati che prende tutti i valori leggendoli dal database e crea un oggetto di tipo Operatore che se passa tutti i controlli richiama il metodo registra_operatore il quale inserisce sul databse tutti i parametri inseriti e l'utente viene rimandato alla pagina "Accedi come operatore".

4. REGISTRAZIONE CENTRO

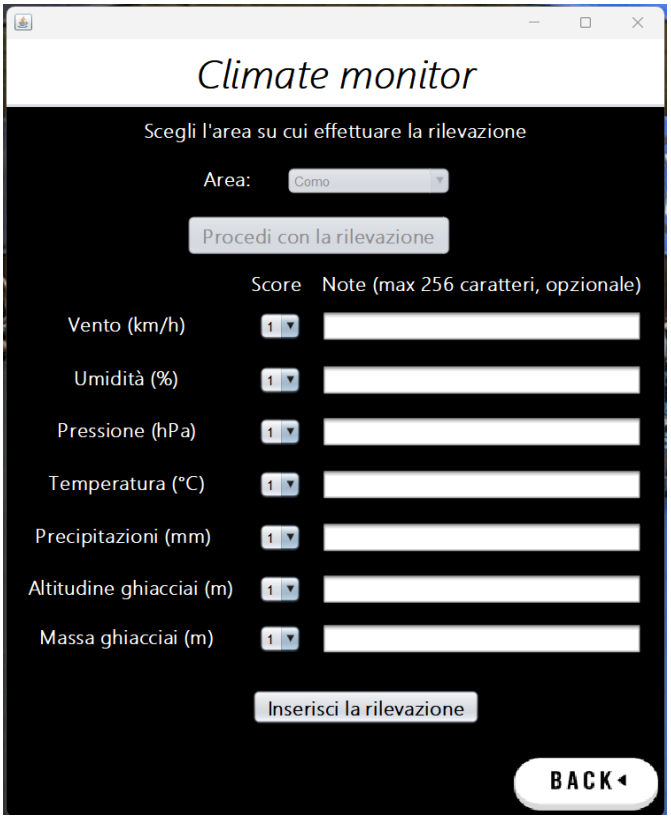
Struttura grafica

Nel caso in cui l'operatore non ha inserito il centro in cui lavora nella fase di registrazione verrà reindirizzato in questa pagina dedicata alla registrazione del centro.
La struttura della pagina è simile a quella della registrazione dell'operatore.
Prima della registrazione del centro però è obbligatorio inserire le aree su cui verranno effettuate le rivelazioni.
Solo dopo aver inserito tutte le "Aree rimanenti" si attiverà il button "Registra centro" che permetterà all'operatore di registrare il proprio centro.

Codice

La classe **RegistraCentro.java** funziona allo stesso modo della classe spiegata precedentemente, dunque è stato creato il metodo controlli (che richiama il metodo resoconto_controlli della classe centro_monitoaggio.java) e il metodo resoconto_controlli_aree (che richiama il metodo controlli_su_area della classe area_interesse.java).
Il tasto "Registra centro" legge i campi dal database e crea il centro di monitoraggio solo se non ci sono errori sui parametri inseriti che nel caso verranno segnalati tramite pop-up.
Nel caso in cui va tutto a buon fine il contenuto dell'oggetto creato viene viene scritto sul database.

5. PARAMETRI



Struttura grafica

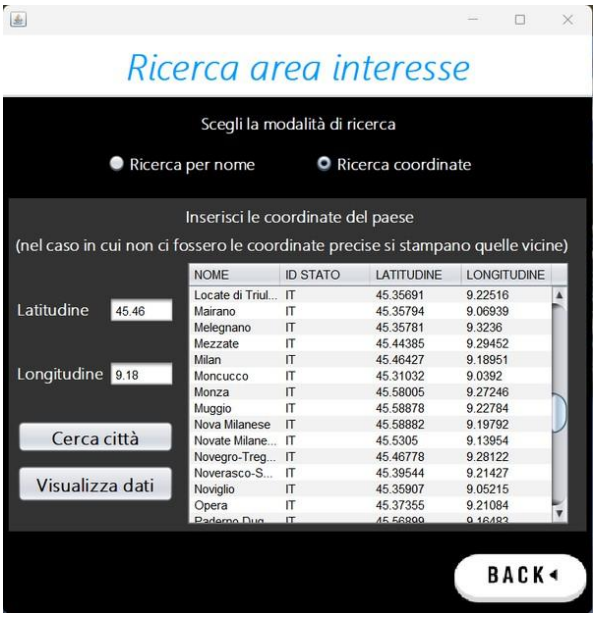
Di fianco alla label "Area" è collocata una combo box contenente le aree

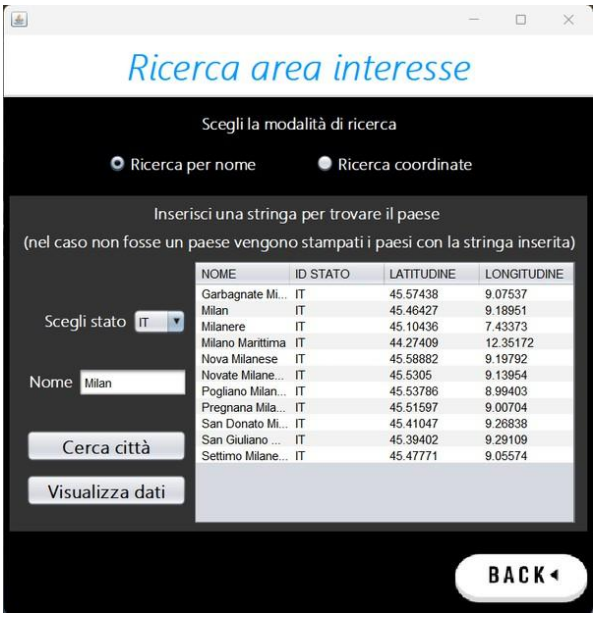
relative all'operatore che ha effettuato il login.
Una volta scelta l'area si può procedere alla rilevazione tramite il button "Procedi alla rilevazione".
Per ciascun parametro è dedicata una combobox con l'intestazione "Score" dove è possibile scegliere un punteggio tra 1 e 5.
Di fianco a ciascuno di questi parametri è inoltre possibile inserire una nota piacimento.
Una volta completata la rilevazione si può inserire all'interno del software tramite il button "Inserisci rilevazioni"

Codice

La classe **InserimentoRilevazione.java** raccoglie le rilevazioni.
Prima di tutto viene eseguita la lettura dal database, quando viene trovato il centro interessato viene suddivisa tutta la stringa di aree in token (tramite lo StringTokinizer) per inizializzare la combo box con le aree di interesse scelte dal centro.
Una volta aver scelto la singola area si può procedere con la rilevazione su di essa che verrà infine inserita nel database per la memorizzazione dei dati.

6. CITTADINO





Struttura grafica

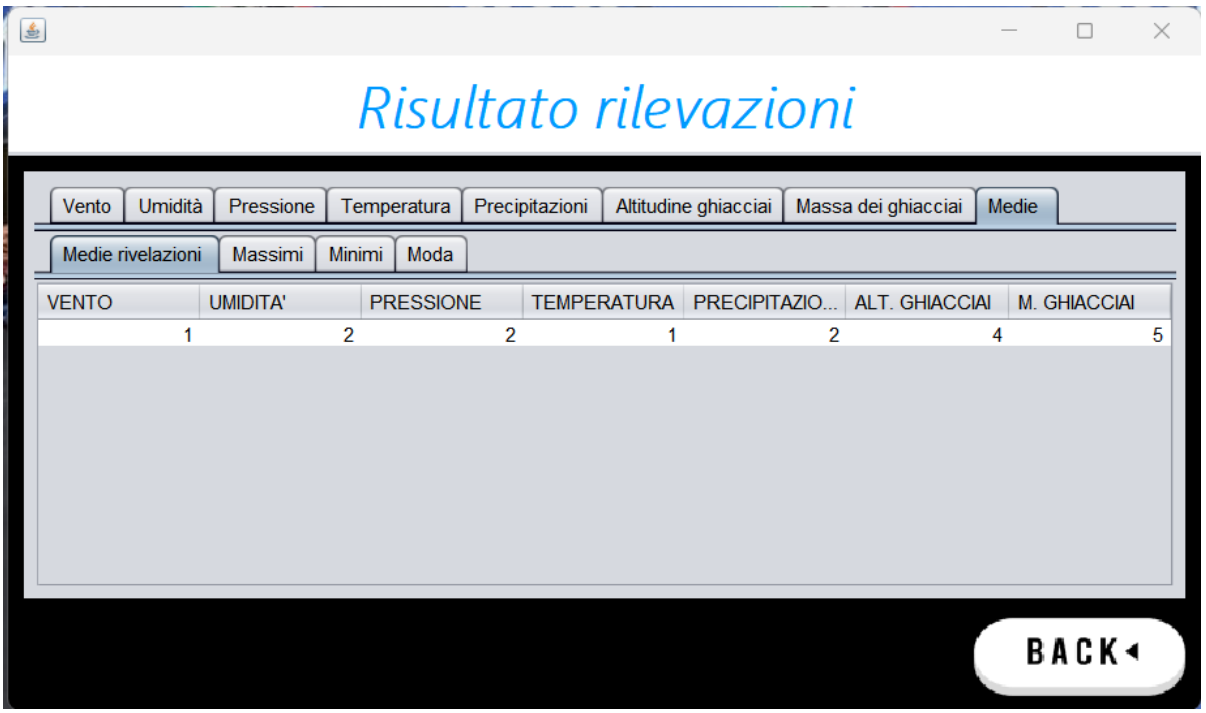
La ricerca dell'area può essere eseguita dal cittadino tramite le coordinate o per nome. Una volta inseriti i dati all'interno delle text fields si può procedere con la ricerca che stamperà sulla tabella i risultati trovati.
Basterà cliccare sulla città desiderata e sul button "Visualizza dati" per visionare le rilevazioni eseguite su quella determinata area.

Codice

La classe accedi_come_cittadino.java è suddivisa in due parti, ovvero la ricerca per coordinate e la ricerca per nome:

1. **Ricerca per coordinate** → vengono inserite le coordinate (latitudine e longitudine) e creato un oggetto di tipi area_interesse. Dopodiché si verifica se è presente un paese con le coordinate inserite (con un raggio di approssimazioni di qualche chilometro su cui può oscillare la latitudine e la longitudine). Una volta trovata la singola area o le aree vengono stampate in tabella. Cliccando su quella desiderata si possono visualizzare le rilevazioni effettuate su di essa.
2. **Ricerca per nome** → questa funziona in modo analogo con la precedente con l'unica differenza che viene scelto lo stato dalla combo box (in modo da filtrare maggiormente i dati), viene inserita la stringa nel text field e viene stampato sulla tabella le città dello stato selezionato che combaciano con la stringa inserita.

7. RISULTATO RILEVAZIONI



Codice

Infine abbiamo la classe dati_area_selezionata.java nella quale viene inserita la tabella con le relative sotto tabelle al suo interno.

Qui avviene la lettura da file per tutte le aree che dispongono di rilevazioni. Per ogni parametro è stata creata una tabella con infine il prospetto riassuntivo che contiene la media, il massimo, il minimo e la moda delle rilevazioni calcolati tramite i metodi dedicati.