VER. 2.0.0
03/07/2024

PROGETTO LABORATORIO B
CLIMATE MONITORING
SVILUPPATO DA: BADROUS GIORGIO, BIAVASCHI RAFFAELE, BONACINA DAVIDE, CASATI SIMONE.

MANUALE MANUALE MANUALE MANUALE TECNICO

INDICE DEI CONTENUTI

INTRODUZIONE

"Climate Monitoring" è un progetto sviluppato in ambito accademico durante il '**Laboratorio A'** ed il '**Laboratorio B**' al fine di dimostrare le capacità acquisite nel corso dei primi due anni di studi. Il progetto consiste in un **sistema di monitoraggio/salvataggio di parametri climatici** fornito da centri di monitoraggio, gestiti da Operatori abilitati e in grado di essere mostrati anche ad utenti comuni.

Il progetto è stato sviluppato in **Java 17** usando l'interfaccia grafica già presente nelle librerie base di Java (**libreria Swing**). E' stato utilizzato per lo sviluppo l'IDE: **NetBeans 17**, con **Maven.**
Come Database è stato scelto **PostgreSQL** (> **v.16.x**).

Il progetto è stato strutturato per **funzionare su più piattaforme possibili** (1), ed è stato testato direttamene su:

- **Windows 10 Pro / Windows 11** (Architettura x64)
- **MacOS** (Architettura x64 e ARM)

La **classi principali** (divise per packing) del progetto sono:

- **Accesso** *(pack. 'access')*
- **Registrazione** *(pack. 'access')*
- **AreaParametri** (pack. 'climatemonitoring')
- **ServerCM** *(server, pack. 'climatemonitoring')*
- **ClientCM** *(client/main, pack. 'climatemonitoring')*
- **ClimateInterface** *(interface, pack. 'climatemonitoring')*
- **DataLabelFormatter** *(pack. 'customzation')*
- **TableColumnAdjuster** *(pack. 'customzation')*
- **AreaInt** *(pack. 'operatoractions')*
- **CentroMonitoraggio** *(pack. 'operatoractions')*
- **Parametri** *(pack. 'operatoractions')*

Note sul progetto

1.Durante la progettazione del software è stato scelto di **non usare una libreria grafica esterna** per alleggerire il carico e permettere a **tutte le piattaforme con Java installato** (e quindi librerie base) di eseguirlo.
2.È stato scelto di **dividere il progetto in più package** per una distinzione migliore tra le varie parti.
3. Vista l'implementazione della tecnologia RMI, si è scelto di dividere il progetto in '**Server**' e '**Client**' con il server che ha come parametri **i valori di collegamento a Postgre impostabili lato utente**, cosi da poter distribuire il tutto su più macchine anche remote.

CLASSI

• Accesso

La classe **Accesso** gestisce l'autenticazione degli utenti (operatori) all'interno dell'applicazione ClimateMonitoring. Fornisce un'interfaccia grafica per l'**inserimento delle credenziali** (username e password) e verifica la loro **validità** confrontandole con i dati memorizzati sul server interfacciandosi con **RMI.**

- **Dipendenze**:
  - **javax.swing**: Libreria per la creazione dell'interfaccia grafica (pulsanti, campi di testo, ecc.).
  - **java.awt**: Libreria per la gestione degli eventi dell'interfaccia utente (click del mouse, pressione dei tasti, ecc.).
  - **java.rmi**: Libreria per la comunicazione remota con il server tramite RMI.
  - **java.util.logging**: Libreria per la registrazione dei messaggi di log.
  - **java.awt.***: per gestire gli eventi.
- **Costruttori:**
  - **Accesso():** Costruttore di base, senza parametri. Inizializza la finestra di dialogo ma non la rende visibile.
  - A**ccesso(ClientCM hh, boolean ck)**: Costruttore principale. Riceve un oggetto ClientCM (la finestra principale dell'applicazione) e un valore booleano che indica se bloccare la finestra sottostante. Inizializza la finestra di dialogo, la posiziona al centro dello schermo e la rende visibile.
- **Metodi:**
  - **initComponents()**: Metodo generato automaticamente da NetBeans per inizializzare i componenti grafici dell'interfaccia.
  - **AccediActionPerformed(ActionEvent evt)**: Gestisce l'evento del clic sul pulsante "Accedi". Verifica se i campi username e password sono stati compilati e, in caso affermativo, chiama il metodo accedi() per eseguire l'autenticazione. **log():** Verifica la presenza delle credenziali (username e password). Se presenti, chiama il metodo accedi(), altrimenti segnala un errore all'utente.
  - **accedi():** Esegue l'autenticazione. Contatta il server RMI per verificare le credenziali inserite dall'utente. Se l'autenticazione ha successo, aggiorna l'interfaccia della finestra principale (ClientCM) per riflettere l'accesso avvenuto (ad esempio, rendendo visibili nuovi pulsanti o modificando le etichette).
  - **passwordFieldKeyPressed(KeyEvent evt), usernameFieldKeyPressed(KeyEvent evt), formKeyPressed(KeyEvent evt), AccediKeyPressed(KeyEvent evt):** Questi metodi gestiscono l'evento della pressione del tasto INVIO nei campi di testo e nei pulsanti. Se il tasto INVIO viene premuto, viene chiamato il metodo log() per eseguire l'autenticazione.
  - **main(String[] args):** Metodo principale. Avvia l'interfaccia utente creando un oggetto Accesso e rendendolo visibile.

CLASSI

• Accesso

Eccezioni Gestite:

- **IOException**: Eccezione generata in caso di problemi di input/output, come la mancanza del file delle credenziali o errori nella lettura/scrittura del file.
- **RemoteException**: Eccezione generata in caso di errori nella comunicazione RMI con il server.
- **NotBoundException**: Eccezione generata se il nome non è stato trovato nel registro RMI.

Note Aggiuntive:

- La classe utilizza RMI per comunicare con un **server remoto** che gestisce l'autenticazione degli utenti.
- La classe interagisce con la classe **ClientCM** per aggiornare l'interfaccia utente dopo l'accesso.
- L'accesso rende visibili, ad accesso consentito e quindi verificato, gli elementi disponibili solo all'utente '**Operatore**' per poter inserire i parametri climatici, creare nuovi centri di interesse o creare nuova aree di interesse.

Esempio di codice

```
public void accedi(){
      try {
         setClient();
         //L'utente esiste
         List<String> userInfo=stub.getUtente(usernameField.getText(), passwordField.getText());
         if(!userInfo.isEmpty()){
            String nome = userInfo.get(0);
            String cognome = userInfo.get(1);
            String codfisc = userInfo.get(2);
            hh.logout.setVisible(true);
            hh.accedi.setVisible(false);
            hh.registrati.setVisible(false);
            hh.addCentro.setVisible(true);
            hh.addParam.setVisible(true);
            hh.addArea.setVisible(true);
            hh.nomeU = nome;
            hh.cogU = cognome;
            hh.codFisc = codfisc;
            hh.newLabel.setText("Benvenuto " + nome + " " + cognome);
            this.dispose();
         } else {
            // Credenziali errate, mostra un messaggio di errore
            JOptionPane.showMessageDialog(null, "Le credenziali sono errate!", "Errore!", JOptionPane.ERROR_MESSAGE);
         }
      } catch (RemoteException e) {
         // Gestisci l'eccezione RMI
         JOptionPane.showMessageDialog(null, "Errore di connessione al server!", "Errore!", JOptionPane.ERROR_MESSAGE);
      } catch (NullPointerException e) {
         JOptionPane.showMessageDialog(null, "Stub del server non collegato!", "Errore", JOptionPane.ERROR_MESSAGE);
      }
   }
```

CLASSI

• Registrazione

La classe **Registrazione** gestisce la **registrazione di nuovi operatori nel sistema** ClimateMonitoring. Fornisce un'interfaccia grafica per l'inserimento dei dati dell'operatore (nome, cognome, codice fiscale, email, username, password e centro di monitoraggio associato). I dati vengono poi inviati al server tramite RMI per la registrazione nel database.

- **Dipendenze:**
  - **javax.swing**: Libreria per la creazione dell'interfaccia grafica (pulsanti, campi di testo, menu a tendina, ecc.).
  - **java.awt**: Libreria per la gestione degli eventi dell'interfaccia utente (click del mouse, ecc.). ·
  - **java.rmi**: Libreria per la comunicazione remota con il server tramite RMI.
  - **java.util:** Libreria per l'utilizzo di strutture dati come ArrayList e List.
  - **java.util.logging**: Libreria per la registrazione dei messaggi di log.
- **Costruttori:**
  - **Registrazione()**: Costruttore di base, senza parametri. Inizializza la finestra di dialogo ma non la rende visibile.
  - **Registrazione(ClientCM reg, boolean ck):** Riceve un oggetto ClientCM (la finestra principale dell'applicazione) e un valore booleano che indica se bloccare la finestra sottostante. Inizializza la finestra di dialogo, la posiziona al centro dello schermo, la rende visibile, e ne impedisce il ridimensionamento. Imposta il titolo della finestra a "Registrati" e popola il menu a tendina (centriDrop) con i nomi dei centri di monitoraggio disponibili dal server.
- **Metodi:**
  - **InitComponents()**: Metodo generato automaticamente da NetBeans per inizializzare i componenti grafici dell'interfaccia.
  - **RegistratiActionPerformed(ActionEvent evt)**: Gestisce l'evento del click sul pulsante 'Registrati'.

    ->Validazione dei Dati:
    - Verifica che tutti i campi obbligatori siano stati compilati (nome, cognome...).

    ->Gestione degli Errori:
    - Se la validazione fallisce, viene visualizzato un messaggio di errore con l'elenco degli elementi mancanti ('Non hai inserito: *elenco*...').
    - Se la validazione ha successo, ma viene alzato un'errore da parte del server RMI, viene visualizzato un messaggio di errore contente l'indicazione all'errore ('Eccezione:...').

Invia i dati dell'operatore al server tramite il metodo registrazione dell'**interfaccia RMI**.

CLASSI

Registrazione

- **centriDropSelector()**: Recupera dal server i nomi dei centri di monitoraggio disponibili e li aggiunge al menu a tendina centriDrop.
- **setClient()**: Inizializza la connessione RMI con il server.

->Gestione degli Errori:

- Utilizza un blocco try-catch per gestire eventuali eccezioni RemoteException o NotBoundException.
- In caso di errore, viene stampato un messaggio di errore sulla console e l'eccezione viene rilanciata.

- **main(String[] args)**: Metodo principale. Avvia l'interfaccia utente creando un oggetto Registrazione e rendendolo visibile.

->Messaggi all'Utente:

Errori:

- 'Dati Mancanti: [elenco dei campi mancanti]'
- 'Errore durante la registrazione: [messaggio dell'eccezione]'
- 'Errore impostando il client RMI: [messaggio dell'eccezione]' (visualizzato sulla console)

Esempio di codice

```
public void reg() throws IOException{

    boolean check = true;
    ArrayList<String> errore = new ArrayList<>();
    int c = 0;

    if (nome.getText().equals("")) { check = false; errore.add("Nome"); c++; }
    if (cognome.getText().equals("")) { check = false; errore.add("Cognome"); c++; }
    if (email.getText().equals("")) { check = false; errore.add("Email"); c++; }
    if (username.getText().equals("")) { check = false; errore.add("Username"); c++; }
    if (codFisc.getText().equals("")) { check = false; errore.add("Codice Fiscale"); c++; }
    if (password.getText().equals("")) { check = false; errore.add("Password"); c++; }
    if (centriDrop.getSelectedItem().equals("")) { check = false; errore.add("Centro Monitoraggio"); c++; }

    if (!check) {
       String f = "";
       for (int i = 0; i < c; i++) {
          f += errore.get(i) + "\n";
       }
       JOptionPane.showMessageDialog(null, "Dati Mancanti : \n" + f, "Error", JOptionPane.ERROR_MESSAGE);
    } else {
       try {
          String centroNome = centriDrop.getSelectedItem().toString();
          stub.registrazione(nome.getText(), cognome.getText(), codFisc.getText(), email.getText(), username.getText(), password.getText(), centroNome);
       } catch (RemoteException e) {
          // Gestione dell'eccezione remota
          Logger.getLogger(Registrazione.class.getName()).log(Level.SEVERE, null, e);
          JOptionPane.showMessageDialog(this, "Errore durante la registrazione: \n" + e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
       } catch(NullPointerException e){
          JOptionPane.showMessageDialog(this, "Stub del server non collegato!", "Errore", JOptionPane.ERROR_MESSAGE);
          dispose();
       }
    }
 }
```

CLASSI

• AreaParametri

La classe **AreaParametri** visualizza i **parametri climatici associati a una specifica area di interesse** all'interno dell'applicazione ClimateMonitoring. Utilizza RMI per recuperare i dati dal server e li presenta in una **tabella organizzata**.

- **Dipendenze:**
  - **javax.swing**: Libreria per la creazione dell'interfaccia grafica (pulsanti, campi di testo, tabelle, ecc.).
  - **java.awt**: Libreria per la gestione degli eventi dell'interfaccia utente (click del mouse, ecc.).
  - **java.rmi**: Libreria per la comunicazione remota con il server tramite RMI.
  - **java.util**: Libreria per l'utilizzo di strutture dati come ArrayList, List, e Map.
  - **java.util.logging**: Libreria per la registrazione dei messaggi di log.

- **Costruttori:**
  - **AreaParametri()**: Costruttore di base, senza parametri. Inizializza la finestra di dialogo ma non la rende visibile.
  - **AreaParametri(ClientCM hh, boolean cs, long geo):** Riceve un oggetto ClientCM (la finestra principale dell'applicazione), un valore booleano che indica se bloccare la finestra sottostante, e il GeoNameID dell'area di interesse. Inizializza la finestra di dialogo, recupera i parametri climatici dal server, e imposta la visibilità della finestra in base alla presenza o meno di dati.

- **Metodi:**
  - **initComponents()**: Metodo generato automaticamente da NetBeans per inizializzare i componenti grafici dell'interfaccia.
  - **paramTableMouseClicked(MouseEvent evt**): Gestisce l'evento di clic del mouse sulla tabella dei parametri. Se l'utente clicca su una riga che contiene note, viene aperta una finestra di dialogo per visualizzare le note associate.
  - **main(String[] args)**: Metodo principale. Avvia l'interfaccia utente creando un oggetto AreaParametri e rendendolo visibile.
  - **visualizzaMedia, visualizzaModa,visualizzaMediana()**: visualizzano ognuno il sunto della media, moda e mediana.
  - **visualizzaParametriClimatici()**: Metodo cheverifica la presenza dei parametri usando il metodo 'visualizzaParametriClimaticiDB' presente su 'ServerCM'

    ->Gestione degli Errori:
    - Utilizza un blocco try-catch per gestire eventuali eccezioni RemoteException o NotBoundException che possono verificarsi durante la comunicazione RMI.
    - In caso di errore, viene stampato un messaggio di errore sulla console e l'eccezione viene rilanciata per segnalare il problema al chiamante.
    - Recupera i parametri climatici dal server tramite RMI (metodo visualizzaParametriClimaticiDB).

CLASSI

- ## AreaParametri

  ->Validazione dei Dati:
  - Verifica se sono stati restituiti dei dati dal server.

  ->Messaggi all'Utente:
  - Se non ci sono dati, viene visualizzato un messaggio di avvertimento ("Non sono disponibili parametri climatici per la seguente città!").
  - Se ci sono dati, popola la tabella con i parametri climatici.

- **addRowTable(String[] dataRow)**: Aggiunge una riga di dati alla tabella dei parametri climatici.
- **clearTable()**: Svuota la tabella dei parametri climatici.
- **mostraFinestraNote(String note)**: Visualizza una finestra di dialogo con le note associate a un parametro climatico.
- **setClient():** Inizializza la connessione RMI con il server.

  ->Gestione degli errori:
  - Utilizza un blocco try-catch per gestire eventuali eccezioni RemoteException o NotBoundException.
  - In caso di errore, viene stampato un messaggio di errore sulla console e l'eccezione viene rilanciata.

  ->Messaggi all'utente:

  Errori:
  - "Errore impostando il client RMI: [messaggio dell'eccezione]" (visualizzato sulla console)

  Avvisi:
  - "Non sono disponibili parametri climatici per la seguente città!"

  Successo:
  - Nessun messaggio specifico di successo viene visualizzato, ma i dati vengono correttamente visualizzati nella tabella in caso di successo.

Ulteriori Note:

La tabella dei parametri climatici è configurata **per essere non modificabile dall'utente** e per avere le celle centrate orizzontalmente.
Le **note sono apribili cliccando su ogni singolo valore climatico**, e sono autonomi tra di loro. Se è presente una nota, per quel determinato valore climatico, il valore è distinto da '**'.
I valori di Media, Moda e Mediana sono **arrotondati per difetto**.

CLASSI

ServerCM

La classe **ServerCM** rappresenta il server del nostro protocollo RMI e **funge da accesso al DB da parte del client**. Contiene i metodi veri e propri che vanno poi ad interagire con il server, inoltre fornisce **un'interfaccia grafica utile all'utente per inserire i parametri di accesso al database Postgre**, oltre che permettere la chiusura della connesione.

- **inserisciCentroMonitoraggio(String nome, String indirizzo, String elencoAree)**:

  Inserisce i parametri climatici nel database, associandoli al centro di monitoraggio e all'area geografica corretti.

  ->Validazione dei Dati:

  - Verifica che tutti i campi obbligatori siano forniti.
  - Se la validazione fallisce, lancia un'eccezione RemoteException con un messaggio di errore dettagliato ("Non hai inserito: [elenco campi]").

  ->Gestione degli Errori:

  - Utilizza un blocco try-catch per catturare le eccezioni SQLException che possono verificarsi durante l'interazione con il database.
  - In caso di errore, registra l'errore nel log e rilancia un'eccezione RemoteException con un messaggio di errore appropriato.

- **inserisciAreaDB(String citta, String code, String country, String lat, String lon)**: Inserisce una nuova area geografica nel database, normalizzando il nome della città in formato ASCII.
- **visualizzaParametriClimaticiDB(String geoNameID)**: Recupera i parametri climatici associati a un'area geografica specifica (identificata dal GeoNameID) dal database.
- **main(String[] args)**: Avvia il server, crea un registro RMI sulla porta 1099 e registra l'oggetto ServerCM con il nome "ClimateMonitoring".
- **getUtente(String username, String password)**: Recupera le informazioni dell'utente (nome, cognome, codice fiscale) in base alle credenziali fornite.
- **getCentriMonitoraggio()**: Restituisce una lista dei nomi di tutti i centri di monitoraggio nel database.
- **getCentriMonitoraggio(String codFisc)**: Restituisce una lista dei nomi dei centri di monitoraggio associati a un operatore specifico (identificato dal codice fiscale).
- **getAreeInteresse(String nomeCentro)**: Restituisce una lista delle aree di interesse associate a un centro di monitoraggio specifico.
- **toAscii(String accented)**: Converte una stringa con caratteri accentati nel suo equivalente ASCII, rimuovendo gli accenti.

CLASSI

- **ServerCM**

->Messaggi di Errore (RemoteException):

- "Dati Mancanti: [elenco campi]" (registrazione operatore)
- "Non hai inserito: [elenco campi]" (inserimento centro di monitoraggio)
- "Errore durante l'inserimento del centro di monitoraggio nel database: [messaggio dell'eccezione]"
- "Errore durante l'inserimento dei dati nel database: [messaggio dell'eccezione]" (inserimento parametri climatici)
- "Errore durante l'inserimento dei dati nel database." (inserimento area geografica)
- "Centro di monitoraggio non trovato!" (registrazione operatore)
- "Database error: [messaggio dell'eccezione]" (varie operazioni)

Esempio di codice

```
/**
    * Metodo per la registrazione dell'utente dati i dati, email, username, password e il centro di competenza
    * @param nome, tipo 'String' è il nome dell'utente
    * @param cognome, tipo 'String' è il cognome dell'utente
    * @param codFisc, tipo 'String' è il codice fiscale dell'utente
    * @param email, tipo 'String' è la email dell'utente
    * @param username, tipo 'String' è lo username dell'utente
    * @param password, tipo 'String' è la password dell'utente
    * @param centroNome, tipo 'String' sono il/i nomi dei centri di comptenza
    * @throws java.rmi.RemoteException
    */
   @Override
   public synchronized void registrazione(String nome, String cognome, String codFisc, String email, String username, String password, String centroNome)
throws RemoteException{
      boolean check = true;
       ArrayList<String> errore = new ArrayList<>();
       int c = 0;
       //CONTROLLO ERRORI
       if (nome.isEmpty()) { check = false; errore.add("Nome"); c++; }
       if (cognome.isEmpty()) { check = false; errore.add("Cognome"); c++; }
       if (email.isEmpty()) { check = false; errore.add("Email"); c++; }
       if (username.isEmpty()) { check = false; errore.add("Username"); c++; }
       if (codFisc.isEmpty()) { check = false; errore.add("Codice Fiscale"); c++; }
       if (password.isEmpty()) { check = false; errore.add("Password"); c++; }
       if (centroNome.isEmpty()) { check = false; errore.add("Centro Monitoraggio"); c++; }
       if (!check) {
          String f = "";
          for (int i = 0; i < c; i++) {
             f += errore.get(i) + "\n";
          }
          // Gestione errori appropriata per l'applicazione server
          throw new RemoteException("Dati Mancanti:\n" + f);

       //DATI CORRETTI
       } else {
          try {
             //dbConnection();
             // Recupero dell'ID del centro monitoraggio
             String getCentroIdSql = "SELECT idcentro FROM centromonitoraggio WHERE nome = ?";
             pstmt = conn.prepareStatement(getCentroIdSql);
             pstmt.setString(1, centroNome);
             rs = pstmt.executeQuery();

             ....
       }
   }
```

CLASSI

ClientCM

La classe **ClientCM** rappresenta la finestra principale dell'applicazione ClimateMonitoring e **funge da client RMI**. Consente all'utente di **cercare aree geografiche per nome o coordinate**, visualizzare i parametri climatici associati e, se **autenticato** come operatore, **inserire o modificare dati** relativi a centri di monitoraggio, aree di interesse e parametri climatici.

- **Dipendenze:**
  - **javax.swing**: Libreria per la creazione dell'interfaccia grafica (pulsanti, campi di testo, tabelle, menu a tendina, ecc.).
  - **java.awt**: Libreria per la gestione degli eventi dell'interfaccia utente e layout.
  - **java.rmi**: Libreria per la comunicazione remota (RMI) con eccezioni.
  - **java.util**: Libreria per l'utilizzo di strutture dati come ArrayList e List.
  - **java.util.logging**: Libreria per la registrazione dei messaggi di log.
- **Costruttore:**
  - **ClientCM():** Inizializza la finestra principale, impostando il titolo, rendendola visibile, disabilitando il ridimensionamento e posizionandola al centro dello schermo. Inizializza i componenti dell'interfaccia grafica e tenta di stabilire una connessione con il server RMI.
- **Metodi:**
  - **initComponents()**: Metodo generato automaticamente per l'inizializzazione dei componenti grafici.
  - **refresh()**: Aggiorna l'interfaccia grafica della finestra.
  - **nomeButtonActionPerformed(ActionEvent evt)**: Gestisce il click del pulsante cerca (per nome).

    ->Validazione dei Dati:
    - Verifica se il campo del nome è vuoto. Se è vuoto, mostra un messaggio di errore

    ->Messaggi all'Utente:
    - "Non hai inserito alcun nome!" (se il campo del nome è vuoto).
  - **coordButtonActionPerformed(ActionEvent evt)**: Gestisce il click del pulsante cerca (per coordinate).

    ->Validazione dei Dati:
    - Verifica se i campi lat. e long. contengono valori non numerici o nulli.

    ->Messaggi all'Utente:
    - "Coordinate non valide!" (se i campi sono vuoti o non contengono numeri).
  - **offsetSlideStateChanged (Changevent evt)**: Aggiorna il valore dell'offset quando lo slider viene 'spostato' (quindi modificato di posizione).

CLASSI

ClientCM

- **accediActionPerformed(ActionEvent evt)**: Apre la finestra di dialogo Accesso per l'autenticazione dell'utente.
- **registratiActionPerformed(ActionEvent evt)**: Apre la finestra di dialogo Registrazione per la registrazione di un nuovo utente.
- **logoutActionPerformed(ActionEvent evt):** Esegue il logout dell'utente, nascondendo i pulsanti specifici per gli operatori e modificando l'etichetta di benvenuto.
- **addCentroActionPerformed(ActionEvent evt)**: Apre la finestra di dialogo CentroMonitoraggio per l'inserimento di un nuovo centro.
- **cancelActionPerformed(ActionEvent evt)**: Pulisce la tabella dei risultati e resetta i campi di input.
- **addParamActionPerformed(ActionEvent evt)**: Apre la finestra di dialogo Parametri per l'inserimento dei parametri climatici.
- **resTableMouseClicked(MouseEvent evt)**: Gestisce il click sulla tabella dei risultati. Apre la finestra AreaParametri per visualizzare i dettagli dell'area selezionata, se l'utente è autenticato.
- **nomeFieldKeyPressed(KeyEvent evt), lonFieldKeyPressed(KeyEvent evt), latFieldKeyPressed(KeyEvent evt), offsetSlideKeyPressed(KeyEvent evt):** Gestiscono la pressione del tasto INVIO nei rispettivi campi di input, eseguendo le azioni corrispondenti (ricerca per nome o coordinate).
- **addAreaActionPerformed(ActionEvent evt)**: Apre la finestra di dialogo AreaInt per l'aggiunta di una nuova area di interesse (solo se l'utente è autenticato come operatore).
- **jFormattedTextField1ActionPerformed(ActionEvent evt)**: Non ha alcuna funzionalità (generato automaticamente).
- **main(String[] args)**: Avvia l'applicazione creando un oggetto ClientCM, impostando il look and feel e stabilendo la connessione RMI.
- **cercaAreaGeografica(String nome)**: Effettua una chiamata RMI al server per cercare aree geografiche in base al nome e popola la tabella dei risultati.
- **cercaAreaGeografica(double lat, double lon, int offset**): Effettua una chiamata RMI al server per cercare aree geografiche in base alle coordinate e all'offset, popola la tabella.
- **addRowTable(String[] dataRow)**: Aggiunge una riga alla tabella dei risultati.
- **clearTable():** Svuota la tabella dei risultati.
- **setClient()**: Stabilisce una connessione con il server RMI.
  ->Messaggi all'Utente:
  Errori:
  - "Non è stato trovato alcun risultato!" (ricerca per nome)
  - "Non sono state trovate aree nelle vicinanze" (ricerca per coordinate)
  - "Coordinate non valide!"
  - "Errore impostando il client RMI: [messaggio dell'eccezione]" (sulla console)

  Successo:
  - "Benvenuto!" (utente non autenticato)
  - "Benvenuto [nome] [cognome]!" (utente autenticato)

CLASSI

Interfaccia "ClimateInterface"

L'interfaccia **ClimateInterface** definisce i **metodi remoti** che un server RMI deve implementare per fornire i servizi dell'applicazione ClimateMonitoring. Questi metodi consentono al client di interagire con il **database** sottostante attraverso la **comunicazione RMI**, **astraendo** i dettagli di implementazione del server.

- Dipendenze:
  - **java.rmi**: Libreria per la comunicazione remota (RMI).
  - **java.sql**: Libreria per l'interazione con il database.
  - **java.util**: Libreria per l'utilizzo di strutture dati come List e Map.
- Metodi Remoti:
  - **dbConnection()**: Stabilisce una connessione al database.
  - **dbDisconnection()**: Chiude la connessione al database.
  - **cercaAreaGeograficaDB(String nome)**: Cerca aree geografiche nel database in base al nome fornito. Restituisce una lista di mappe, dove ogni mappa rappresenta una riga di risultati contenente informazioni come GeoNameID, nome, nome del paese e codice del paese.
  - **cercaAreaGeograficaDB(double lat, double lon, int offset)**: Cerca aree geografiche nel database in base alle coordinate geografiche (latitudine e longitudine) e a un offset specificato. Restituisce una lista di mappe come il metodo precedente.
  - **getUtente(String username, String password)**: Recupera i dati dell'utente (nome, cognome, codice fiscale) in base alle credenziali fornite.
  - **registrazione(String nome, String cognome, ..., String centroNome)**: Registra un nuovo operatore nel sistema, associandolo a un centro di monitoraggio.
  - **getCentriMonitoraggio()**: Restituisce una lista dei nomi di tutti i centri di monitoraggio disponibili nel sistema.
  - **inserisciCentroMonitoraggio(String nome, String indirizzo, String elencoAree)**: Inserisce un nuovo centro di monitoraggio nel database con il nome, l'indirizzo e un elenco di aree di interesse associate.
  - **inserisciParametriClimatici(String nomeCentro, String nomeArea, ...,)**: Inserisce i parametri climatici per una specifica area di interesse e centro di monitoraggio, includendo vento, umidità, pressione, temperatura, precipitazioni, altitudine dei ghiacciai, massa dei ghiacciai e note opzionali.
  - **inserisciAreaDB(String citta, String code, String country, String lat, String lon)**: Inserisce una nuova area di interesse nel database, specificando la città, il codice del paese, il nome del paese e le coordinate geografiche.
  - **visualizzaParametriClimaticiDB(String geoNameID)**: Recupera i parametri climatici associati a un'area geografica specifica (identificata dal GeoNameID).
  - **getCentriMonitoraggio(String codFisc)**: Restituisce una lista dei nomi dei centri di monitoraggio ai quali un operatore (identificato dal codice fiscale) è autorizzato ad accedere.
  - **getAreeInteresse(String nomeCentro)**: Restituisce una lista delle aree di interesse associate a un centro di monitoraggio specifico.

CLASSI

• Interfaccia “ClimateInterface”

->Eccezioni:

- **RemoteException**: Può essere lanciata da tutti i metodi per indicare un errore durante la comunicazione RMI.
- **SQLException**: Può essere lanciata dai metodi che interagiscono con il database per indicare un errore di accesso o esecuzione di query SQL.

->Implementazione Server:

- La **classe ServerCM deve implementare questa interfaccia**, fornendo il codice effettivo per eseguire le operazioni sul database.
- **Binding RMI**: Il server deve esporre un'istanza di ServerCM tramite RMI, associandola a un nome (ad esempio, "ClimateMonitoring") per consentire al client di localizzarla.
- **Client RMI**: La classe ClientCM (o altre classi client) può utilizzare questa interfaccia per interagire con il server remoto e richiedere l'esecuzione dei metodi definiti.

Esempio di codice

```
/**
    * Effettua la connessione al database.
    *
    * @throws SQLException Se si verifica un errore di accesso al database.
    * @throws RemoteException Se si verifica un errore di comunicazione RMI.
    */
   void dbConnection() throws SQLException, RemoteException;

   /**
    * Effettua la disconnessione dal database.
    *
    * @throws RemoteException Se si verifica un errore di comunicazione RMI.
    * @throws SQLException Se si verifica un errore di accesso al database.
    */
   void dbDisconnection() throws RemoteException, SQLException;

   /**
    * Cerca un'area geografica nel database in base al nome.
    *
    * @param nome Il nome dell'area geografica da cercare.
    * @return Una lista di mappe con i dettagli dell'area geografica trovata.
    * @throws RemoteException Se si verifica un errore di comunicazione RMI.
    */
   List<Map<String, String>> cercaAreaGeograficaDB(String nome) throws RemoteException;

/**
    * Cerca un'area geografica nel database in base alle coordinate.
    *
    * @param lat La latitudine dell'area.
    * @param lon La longitudine dell'area.
    * @param offset L'offset per la ricerca.
    * @return Una lista di mappe con i dettagli dell'area geografica trovata.
    * @throws RemoteException Se si verifica un errore di comunicazione RMI.
    */
   List<Map<String, String>> cercaAreaGeograficaDB(double lat, double lon, int offset) throws RemoteException;
```

CLASSI

AreaInt

La classe **AreaInt** gestisce l'inserimento di **nuove aree di interesse** all'interno dell'applicazione ClimateMonitoring. Fornisce un'interfaccia grafica per l'inserimento dei dati dell'area (città, sigla stato, nome stato, latitudine e longitudine) e li invia al server tramite RMI per la **memorizzazione nel database**.

- Dipendenze:
  - **javax.swing**: Libreria per la creazione dell'interfaccia grafica (pulsanti, campi di testo, ecc.).
  - **java.awt**: Libreria per la gestione degli eventi dell'interfaccia utente (click del mouse, ecc.).
  - **java.rmi**: Libreria per la comunicazione remota con il server tramite RMI.
  - **java.util**: Libreria per l'utilizzo di strutture dati come ArrayList.
  - **java.util.logging**: Libreria per la registrazione dei messaggi di log.
- Costruttori:
  - **AreaInt(ClientCM hh, boolean ck)**: Costruttore principale. Riceve un oggetto ClientCM (la finestra principale dell'applicazione) e un valore booleano che indica se bloccare la finestra sottostante. Inizializza la finestra di dialogo, la posiziona al centro dello schermo, la rende visibile e ne impedisce il ridimensionamento.
- Metodi:
  - **initComponents()**: Metodo generato automaticamente da NetBeans per inizializzare i componenti grafici dell'interfaccia.
  - **inserisciActionPerformed(ActionEvent evt)**: Gestisce l'evento del clic sul pulsante "Inserisci"

    ->Validazione dei Dati:

    - Verifica che tutti i campi di testo siano stati compilati.
    - Controlla che i valori di latitudine e longitudine siano numeri validi (utilizzando Double.valueOf()).

    ->Gestione degli Errori:

    - Se la validazione fallisce, viene visualizzato un messaggio di errore specifico (ad esempio, "Non hai inserito: Nome Centro Monitoraggio, Indirizzo fisico, ecc." oppure "Inserisci coordinate corrette!").
    - Se la validazione ha successo, chiama il metodo inserisciAreaInt() per inviare i dati al server.

CLASSI

- **AreaInt**
  - **inserisciAreaInt()**:
    Stabilisce una connessione con il server RMI e chiama il metodo remoto inserisciAreaDB per memorizzare i dati dell'area nel database.

    -> Gestione degli Errori:

    - Utilizza un blocco try-catch per gestire eventuali eccezioni RemoteException o NotBoundException che possono verificarsi durante la comunicazione RMI.
    - In caso di errore, viene visualizzato un messaggio di errore generico ("Errore durante l'inserimento dei dati").

    ->Messaggi all'Utente:

    - Se l'inserimento ha successo, viene visualizzato un messaggio di conferma ("Dati inseriti con successo!").
  - **setClient()**:
    Inizializza la connessione RMI con il server.

    -> Gestione degli Errori:

    - Utilizza un blocco try-catch per gestire eventuali eccezioni RemoteException o NotBoundException.
    - In caso di errore, viene stampato un messaggio di errore sulla console e l'eccezione viene rilanciata per segnalare il problema al chiamante.
  - **main(String[] args)**: Metodo principale. Avvia l'interfaccia utente creando un oggetto AreaInt e rendendolo visibile.
  - Eccezioni Gestite:
    - **IOException**: Eccezione generata in caso di problemi di input/output durante la comunicazione RMI.
    - **RemoteException**: Eccezione generata in caso di errori nella comunicazione RMI con il server.
    - **NotBoundException**: Eccezione generata se il nome del servizio RMI ("ClimateMonitoring") non è stato trovato nel registro RMI.
    - **NumberFormatException**: Eccezione generata se i valori inseriti nei campi latitudine e longitudine non sono numeri validi.
  - Messaggi all'utente:
    Errori:
    - "Non hai inserito: [elenco dei campi mancanti]"
    - "Inserisci coordinate corrette!"

CLASSI

CentroMonitoraggio

La classe **CentroMonitoraggio** gestisce l'**inserimento** di **nuovi centri di monitoraggio climatico** nell'applicazione ClimateMonitoring. Fornisce un'interfaccia grafica per l'inserimento del nome del centro, l'indirizzo fisico e l'elenco delle aree di interesse associate. I dati vengono poi inviati al server tramite RMI per la memorizzazione.

- Dipendenze:
  - **javax.swing**: Libreria per la creazione dell'interfaccia grafica (pulsanti, campi di testo, ecc.).
  - **java.awt**: Libreria per la gestione degli eventi dell'interfaccia utente (click del mouse, ecc.).
  - **java.rmi**: Libreria per la comunicazione remota con il server tramite RMI.
  - **java.util**: Libreria per l'utilizzo di strutture dati come ArrayList.
  - **java.util.logging**: Libreria per la registrazione dei messaggi di log.
- Costruttori:
  - **CentroMonitoraggio(ClientCM hh, boolean ck)**: Riceve un oggetto ClientCM (la finestra principale dell'applicazione) e un valore booleano che indica se bloccare la finestra sottostante. Inizializza la finestra di dialogo, la posiziona al centro dello schermo, la rende visibile, e ne impedisce il ridimensionamento. Imposta il titolo della finestra a "Inserisci Centro" e stabilisce la connessione RMI con il server.
- Metodi:
  - **initComponents()**: Metodo generato automaticamente da NetBeans per inizializzare i componenti grafici dell'interfaccia.
  - **inserisciActionPerformed(ActionEvent evt**): Gestisce l'evento del clic sul pulsante "Inserisci".

    ->Validazione dei Dati:
    - Verifica che tutti i campi obbligatori siano stati compilati (nome centro, indirizzo, elenco aree).

    ->Gestione degli Errori:
    - Se la validazione fallisce, viene visualizzato un messaggio di errore con l'elenco dei campi mancanti ("Non hai inserito: [elenco dei campi mancanti]").
    - Se la validazione ha successo, chiama il metodo registraCentroAree() per inviare i dati al server.

CLASSI

CentroMonitoraggio

- **registraCentroAree()**:
  ->Gestione degli Errori:
  - Utilizza un blocco try-catch per catturare eventuali eccezioni RemoteException durante la comunicazione RMI.
  - In caso di errore, viene visualizzato un messaggio di errore generico con il messaggio dell'eccezione.
  - Invia i dati del centro di monitoraggio al server tramite il metodo inserisciCentroMonitoraggio dell'interfaccia RMI.

  ->Messaggio dell'utente:
  - Se l'inserimento ha successo, viene visualizzato un messaggio di conferma ("Centro di monitoraggio inserito con successo!").
- **setClient()**: Inizializza la connessione RMI con il server.
  ->Gestione degli Errori:
  - Utilizza un blocco try-catch per gestire eventuali eccezioni RemoteException o NotBoundException.
  - In caso di errore, viene stampato un messaggio di errore sulla console e l'eccezione viene rilanciata.
- **main(String[] args)**: Metodo principale, avvia l'interfaccia utente creando un oggetto CentroMonitoraggio e rendendolo visibile.
  ->Messaggio dell'utente:

  Errori:
  - "Non hai inserito: [elenco dei campi mancanti]"
  - "[Messaggio dell'eccezione RemoteException]"

  Successo:
  - "Centro di monitoraggio inserito con successo!"

CLASSI

Parametri

La classe **Parametri** gestisce l'inserimento di **parametri climatici per specifiche aree di interesse** all'interno dell'applicazione ClimateMonitoring. Fornisce un'interfaccia grafica per selezionare il centro di monitoraggio e l'area, e inserire i valori dei parametri (**vento, umidità, pressione, temperatura, precipitazioni, altitudine ghiacciai, massa ghiacciai) e eventuali note.** I dati sono poi inviati al server tramite RMI.

- Dipendenze:
  - **javax.swing**: Per la creazione dell'interfaccia grafica (pulsanti, campi di testo, menu a tendina, ecc.).
  - **java.awt**: Per la gestione degli eventi dell'interfaccia utente.
  - **java.rmi**: Per la comunicazione remota con il server tramite RMI.
  - **java.util**: Per l'utilizzo di strutture dati come ArrayList e List.
  - **java.util.logging**: Per la registrazione dei messaggi di log.
- Costruttori:
  - **Parametri()**: Costruttore di base, inizializza la finestra di dialogo.
  - **Parametri(ClientCM reg, boolean ck**): Riceve un oggetto ClientCM (la finestra principale dell'applicazione) e un valore booleano che indica se bloccare la finestra sottostante. Inizializza la finestra di dialogo, imposta il titolo, recupera la connessione RMI, e popola i menu a tendina con i centri di monitoraggio e le aree di interesse.
- Metodi:
  - **initComponents()**: Metodo generato automaticamente per l'inizializzazione dei componenti grafici.
  - **inserisciActionPerformed(ActionEvent evt)**: Gestisce il click del pulsante "Inserisci", chiamando inserisciParametriClimatici().
  - **inserisciParametriClimatici()**:
    ->Validazione dei Dati:
    - Verifica che tutti i campi obbligatori (centri, aree, parametri climatici) siano compilati.
    - Controlla che i valori dei parametri siano interi validi.
    - Se la validazione fallisce, mostra un messaggio d'errore con i campi mancanti.

    ->Gestione degli Errori:
    - Utilizza un blocco try-catch per catturare RemoteException. In caso d'errore, registra l'errore nel log.
    - Invia i dati al server tramite il metodo inserisciParametriClimatici dell'interfaccia RMI.

    ->Messaggio dell'utente:
    - Mostra un messaggio di successo o di errore a seconda dell'esito dell'inserimento.

CLASSI

Parametri

- **calcolaScore[Parametro](int parametro)**: (Dove [Parametro] è Vento, Umidita, Pressione, Temperatura, Precipitazioni, AltitudineGhiacciai, MassaGhiacciai):
  ->Validazione dei Dati:
  - Verifica che il valore inserito per il parametro sia all'interno dell'intervallo atteso.

  ->Messaggio dell'utente:
  - Mostra un messaggio di errore se il valore non è valido.
  - Calcola e restituisce uno score per il parametro, in base a delle soglie predefinite.
- **centroANDareaDropInitialize()**:
  Popola il menu a tendina centriDrop con i centri di monitoraggio disponibili per l'utente corrente.
  Aggiunge un listener che aggiorna il menu a tendina areaDrop con le aree di interesse associate al centro selezionato in centriDrop.
- **setClient()**: Inizializza la connessione RMI con il server.
  ->Gestione degli Errori:
  - Utilizza un blocco try-catch per gestire eventuali eccezioni RemoteException o NotBoundException.
  - In caso di errore, viene stampato un messaggio di errore sulla console e l'eccezione viene rilanciata.

  ->Messaggio dell'utente:

  Errori:
  - "Non hai inserito: [elenco dei campi mancanti]"
  - "Valore inserito per [parametro] non valido!"
  - "Errore impostando il client RMI: [messaggio dell'eccezione]" (sulla console)

  Successo:
  - "Inserimento effettuato con successo!"

SCELTE ALGORITMICHE

Il sistema **ClimateMonitoring** è stato progettato per fornire una soluzione **efficiente e scalabile** per la gestione e l'analisi dei dati climatici. Le scelte algoritmiche adottate sono state guidate da considerazioni di performance, manutenibilità e adattabilità alle esigenze del sistema.

1. **Algoritmi di Ricerca:**

- **Ricerca per Nome:**
  - Input: Nome dell'area geografica (stringa).
  - Output: Lista di aree geografiche corrispondenti (potenzialmente vuota).
    i. Algoritmo:Connessione al database.
    ii. Esecuzione di una query SQL che utilizza l'operatore LIKE per trovare le aree geografiche il cui nome contiene la stringa di ricerca (ignorando la distinzione tra maiuscole e minuscole).
    iii. Iterazione sui risultati della query e creazione di una lista di mappe, dove ogni mappa rappresenta un'area geografica con i suoi attributi (GeoNameID, nome, nome del paese, codice del paese).
    iv. Restituzione della lista di risultati

  La complessità temporale di questa ricerca dipende dall'implementazione della query SQL nel database. In generale, se la colonna Name è indicizzata, la ricerca può avere una complessità logaritmica (O(log n)), dove n è il numero di aree geografiche nel database. In assenza di indici, la complessità diventa lineare (O(n)).

- **Ricerca per Coordinate**:
  - Input: Latitudine, longitudine (numeri in virgola mobile) e offset (intero).
  - Output: Lista di aree geografiche entro l'offset specificato (potenzialmente vuota).
    i. Algoritmo:Connessione al database.
    ii. Calcolo dell'intervallo di latitudine e longitudine in base all'offset fornito.
    iii. Esecuzione di una query SQL che filtra le aree geografiche in base all'intervallo di coordinate calcolato.
    iv. Iterazione sui risultati della query e creazione di una lista di mappe, come nella ricerca per nome.
    v. Restituzione della lista di risultati.

  La complessità temporale di questa ricerca dipende dall'implementazione della query SQL nel database. In generale, se la colonna Name è indicizzata, la ricerca può avere una complessità logaritmica (O(log n)), dove n è il numero di aree geografiche nel database. In assenza di indici, la complessità diventa lineare (O(n)).

SCELTE ALGORITMICHE

2. **Algoritmi di Validazione:**
   - **Validazione dei Dati di Registrazione:**
     - Input: Dati dell'operatore (nome, cognome, codice fiscale, email, username, password, centro di monitoraggio).
     - Output: Nessuno (lancia un'eccezione se i dati non sono validi).
       i. Algoritmo:Verifica se tutti i campi obbligatori sono stati compilati.
       ii. Se manca qualche campo, crea un messaggio di errore elencando i campi mancanti e lancia un'eccezione RemoteException.
   - **Validazione dei Dati del Centro di Monitoraggio:**
     - Input: Dati del centro di monitoraggio (nome, indirizzo, elenco aree di interesse).
     - Output: Nessuno (lancia un'eccezione se i dati non sono validi).
       i. Algoritmo:Verifica se tutti i campi obbligatori sono stati compilati.
       ii. Se manca qualche campo, crea un messaggio di errore elencando i campi mancanti e lancia un'eccezione RemoteException.
   - **Validazione dei Parametri Climatici:**
     - Input: Parametri climatici (vento, umidità, pressione, ecc.).
     - Output: Nessuno (lancia un'eccezione se i dati non sono validi).
       i. Algoritmo:Verifica se tutti i campi obbligatori sono stati compilati.
       ii. Verifica se i valori dei parametri climatici sono numeri interi validi.
       iii. Se la validazione fallisce, mostra un messaggio di errore all'utente.

   Questi algoritmi hanno una complessità lineare (O(n)), dove n è il numero di campi da validare. Questo perché iterano su tutti i campi per verificarne la presenza e il formato.

3. **Algoritmi di Calcolo degli Score:**
   - **Calcolo dello Score** del Vento, Umidità, Pressione, Temperatura, Precipitazioni, Altitudine Ghiacciai, Massa Ghiacciai:
     - Input: Valore del parametro climatico (intero).
     - Output: Score del parametro (intero da 1 a 5).
       i. Algoritmo: Verifica se il valore del parametro rientra nell'intervallo valido.
       ii. Se il valore è valido, utilizza una serie di istruzioni if-else per determinare lo score in base a soglie predefinite.
       iii. Se il valore non è valido, mostra un messaggio di errore all'utente.
       iv. Restituisce lo score calcolato.

   Questi algoritmi hanno una complessità costante (O(1)), poiché il numero di operazioni da eseguire è fisso e non dipende dalla dimensione dell'input. Le istruzioni if-else vengono eseguite in un numero costante di passi.

SCELTE ALGORITMICHE

ESEMPIO:
Questo algoritmo è implementato nella funzione calcolaScoreVento() della classe Parametri. L'obiettivo è assegnare un punteggio (score) alla velocità del vento, che va da 1 (vento debole) a 5 (vento molto forte). Questo score può essere utilizzato per valutare il rischio associato al vento in una determinata area geografica.

```
public static int calcolaScoreVento(int vento) {
    int score_vento=0;
    if (vento >= 1 && vento <= 10) {
        score_vento=1;
    } else if (vento <= 20) {
        score_vento=2;
    } else if (vento <= 30) {
        score_vento=3;
    } else if (vento <= 60) {
        score_vento=4;
    } else if (vento <= 120) {
        score_vento=5;
    } else {
        JOptionPane.showMessageDialog(null, "Valore inserito per il vento non valido!","Errore!", JOptionPane.ERROR_MESSAGE);
    }
    return score_vento;
}
/**
 * Metodo per il calcolo del punteggio dell'umidita
 * @param umidita intero, valore dell'umidita
 * @return valore score_umidita
```

Spiegazione:

- **Input:** La funzione riceve in input un valore intero vento, che rappresenta la velocità del vento in km/h.
- **Validazione:** Il codice verifica se il valore vento rientra nell'intervallo valido (da 1 a 120 km/h). Se il valore è al di fuori di questo intervallo, viene visualizzato un messaggio di errore all'utente.
- **Calcolo dello Score**: Il codice utilizza una serie di istruzioni if-else per determinare lo score del vento in base a soglie predefinite:
  1: vento tra 1 e 10 km/h
  2: vento tra 11 e 20 km/h
  3: vento tra 21 e 30 km/h
  4: vento tra 31 e 60 km/h
  5: vento tra 61 e 120 km/h
- **Output**: La funzione restituisce il valore score_vento calcolato.

GUI

La GUI (Graphical User Interface) del progetto "Climate Monitoring" è progettata per fornire un'interfaccia utente interattiva e visivamente accattivante che facilita il monitoraggio e la gestione dei dati climatici. La GUI è costruita utilizzando il framework Swing di Java, che offre una vasta gamma di componenti per la creazione di applicazioni desktop. Ecco alcuni dettagli **chiave** sulla GUI di questo progetto:

1. Struttura e Layout

- Multi-finestra: Il progetto utilizza diverse finestre (JDialog e JFrame) per separare le funzionalità, come la registrazione degli utenti, l'accesso, la visualizzazione e l'inserimento dei dati climatici.
- Organizzazione chiara: Ogni finestra ha sezioni ben definite per l'inserimento dei dati, la visualizzazione dei risultati e la navigazione, migliorando l'usabilità per l'utente.

2. Componenti Chiave

- JTable: Utilizzato per visualizzare i dati climatici e altre informazioni in forma tabellare, permettendo agli utenti di vedere chiaramente vari parametri in una sola vista.
- JComboBox: Usato per selezionare opzioni da un elenco dropdown, come nella selezione di aree geografiche o centri di monitoraggio, facilitando l'interazione con il database.
- JButton: Pulsanti per eseguire azioni come la ricerca, l'inserimento dei dati, e il logout.
- JTextField e JPasswordField: Per l'inserimento di testo, come nome utente, password, e parametri di ricerca.
- JLabel: Etichette che forniscono informazioni o istruzioni all'utente, migliorando la comprensione delle varie sezioni dell'interfaccia.

3. Interattività

- Eventi e Listener: La GUI reagisce agli input degli utenti attraverso l'uso di listener per eventi, come azioni su pulsanti o selezioni da combobox, permettendo una dinamica interazione con il database.
- Feedback visivo: L'interfaccia fornisce risposte immediate alle azioni degli utenti, come conferme di successo, avvisi di errori, o aggiornamenti dei dati visualizzati.

4. Accessibilità e Usabilità

- Design intuitivo: La GUI è progettata per essere intuitiva, con controlli facilmente riconoscibili e un layout logico che segue le convenzioni comuni di UI design.
- Gestione degli errori: La GUI gestisce gli errori in modo efficace, mostrando messaggi di errore che aiutano gli utenti a risolvere i problemi durante l'interazione.

5. Estetica e Personalizzazione

- Stile e Tema: L'interfaccia utilizza un tema coeso che è piacevole esteticamente e offre una consistente esperienza utente attraverso le diverse finestre e pannelli.
- Icone e Colori: Utilizza icone descrittive e un schema di colori ben scelto per migliorare la navigazione e l'esperienza complessiva.

6. Scalabilità

- Modularità: La GUI è costruita in modo modulare, permettendo facile estensione o modifica di componenti specifici senza influenzare il resto dell'interfaccia.

MAVEN E FILE POM

Apache Maven è uno strumento di **automazione della build** per progetti Java. Utilizza un approccio di configurazione basato su convenzioni, semplificando la gestione delle dipendenze, il processo di build e la documentazione.

Per il progetto **ClimateMonitoring**, Maven svolge un ruolo centrale nella gestione e standardizzazione del ciclo di sviluppo software. Le funzionalità principali utilizzate nel progetto includono:

- **Gestione delle Dipendenze**: Maven automaticamente risolve e scarica le dipendenze necessarie, come specificato nel file POM. Questo riduce il rischio di conflitti tra librerie e semplifica la configurazione dell'ambiente di sviluppo.
- **Automazione della Build:** Maven compila il codice sorgente, esegue test, e genera pacchetti distribuibili (come JAR files) utilizzando configurazioni predefinite, garantendo coerenza e efficienza.

Project Object Model (POM)

Il Project Object Model (POM) in Maven è un file XML che descrive il progetto software, incluse configurazioni come dipendenze, plugin, versioni e altre impostazioni specifiche. Il POM è essenziale per definire come Maven dovrebbe gestire il progetto.

Il file POM del progetto ClimateMonitoring include diverse sezioni **chiave:**

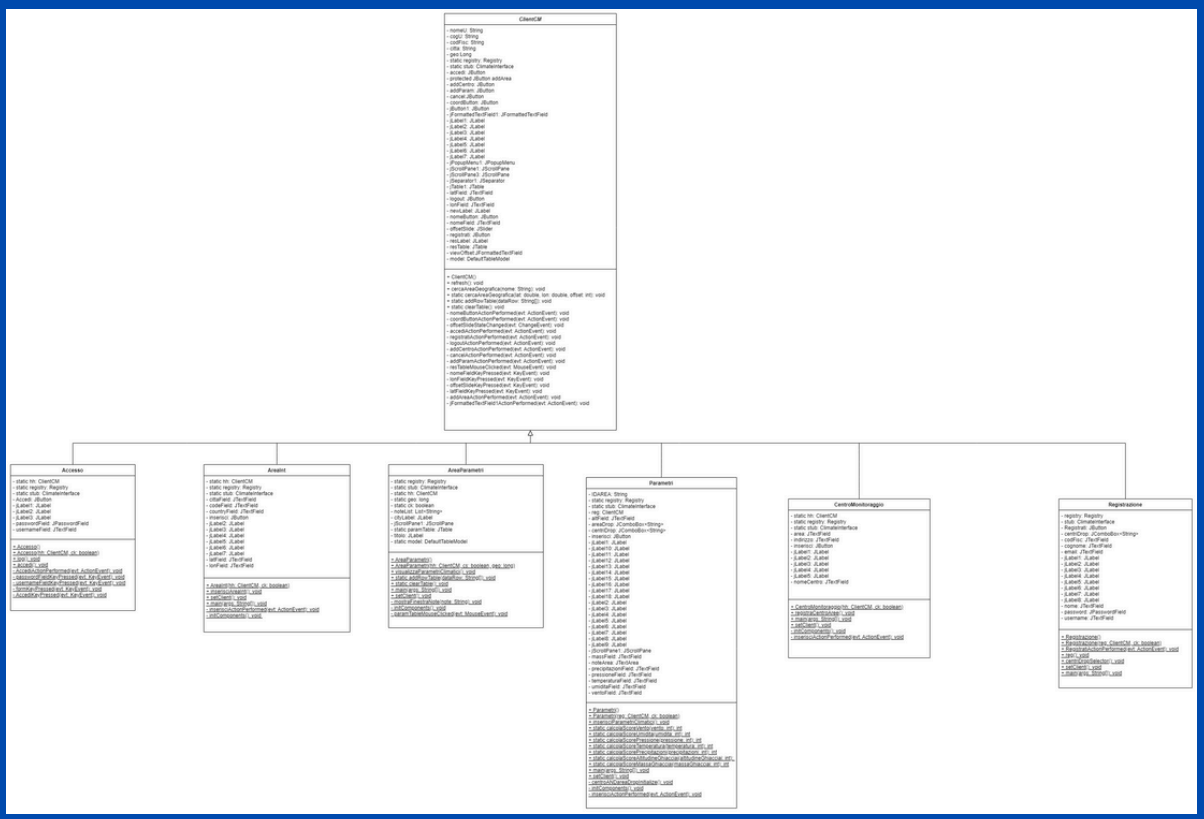
- **Informazioni del Progetto:** Definisce identificativi unici (groupId, artifactId) e la versione del progetto.
- **Proprietà del Compilatore**: Specifica la versione di Java (Java 1.8) per garantire compatibilità e prestazioni ottimali.
- **Dipendenze:** Includi il driver JDBC per PostgreSQL, essenziale per la connessione al database.
- **Plugins:**
  1. Maven Compiler Plugin: Assicura che il progetto sia compilato con la versione corretta di Java.
  2. Exec Maven Plugin: Configura l'esecuzione delle classi principali per il server e il client, semplificando lo sviluppo e il testing.

UML

Per il nostro progetto è stato necessario partire da un'analisi dei requisiti, una suddivisione quindi in una struttura corretta del programma passando **per l'utilizzo dei diagrammi UML**..

A livello strutturale il nostro progetto prevende **un'architettura RMI** in cui un 'Client' o più posso accedere all'informazioni contenuto all'interno di un 'Server', sfruttando appunto il protocollo RMI.
Per procedere quindi con la creazione dei diversi componenti del progetto, e le diverse classi, siamo partiti con il creare un diagramma delle classi che ne rappresentasse le entità, gli attributi necessari e i metodi.
Di seguito ci sono allegati i diagrammi UML delle classi '**ClientCM**' e '**ServerCM**'.

'**Client**' : Diagramma delle classi che lo compongono.



Classi Principali:

- **ClientCM** (Client RMI)
- **Accesso** (Classe per eseguire l'accesso dell'Operatore)
- **AreaInt** (Classe per inserire nuove aree d'interesse, solo Operatore)
- **AreaParametri** (Classe per visualizzare i parametri climatici della area scelta)
- **Parametri** (Classe per inserire nuovi parametri climatici, solo Operatore)
- **CentroMonitoraggio** (Classe per inserire nuovi centri di monitoraggio, solo Operatore)
- **Registrazione** (Classe per registrare nuovi operatori)

Relazioni:

- La classe **ClientC#** è la classe principale e viene utilizzata dalle altre classi
- Le classi **Accesso**, **AreaInt**, **AreaParametri**, **Parametri**,**CentroMonitoraggio** e **Registrazione** utilizzano l'istanza di ClientC# per eseguire varie operazioni specifiche

UML

'**Server**' : Diagramma delle classi che lo compongono.

Classi Principali:

- **ClimateInterface**(Intefaccia che definisce i metodi che vengono implementati dal 'ServerCM')

  Metodi:

  - **dbConnection()** : Esegue la connessione al DB
  - **dbDisconnection()** : Esegue la disconnessione al DB
  - **cercaAreaGeograficaDB**(nome : String): Cerca le aree geografiche dato il nome
  - **cercaAreaGeograficaDB**(lat : double, lon : double, offset : int) Cerca le aree geografiche date le coordinate
  - **getUtente**(username : String, password : String) : Recupera l'utente per il login
  - **registrazione**(nome : String, cognome : String, codiceFisc : String, email : String, username : String, password : String, centroNome : String) : Esegue la registrazione dell'utente
  - **getCentriMontaggio**() : Recupera il centro di monitoraggio.
  - **inserisciCentroMontaggio**(nome : String, indirizzo : String, citta : String) : Inserisce nuovi centri di monitoraggio

UML

- **inserisciParametroClimatico**(nomeCentro : String, nomeArea : String, vento : int, umidita : int, pressione : int, temperatura : int, precipitazioni : int, alt : int, mass : int, note : String) : Inserisce i parametri climatici in base all'area d'interesse scelta.
- **inserisciAreaDB**(citta : String, codice : String, country : String, lat : String, lon : String) : Inserisce nuove aree d'interesse
- **visualizzaParametriClimaticiDB**(nomeArea : String) : Visualizza i parametri climatici di una determinata area d'interesse scelta.
- **getCentriMontaggio(codFisc : String)** : restituisce i centri di monitoraggio
- **getAreeInteresse**(nomeCentro : String) : restituisce le aree d'interesse

- **ServerCM**(classe che implementa l'interfaccia 'ClimateInterface' e fornisce le definizioni concrete per tutti i metodi dichiarati)

  Attributi:

  - **DB_URL** : String -> Indirizzo del server Postgre
  - **DB_USER** : String -> Username di accesso del server Postgre
  - **DB_PASS** : String -> Password di accesso del server Postgre
  - **conn** : Connection -> oggetto Connection per gestire la connessione al database
  - **pstmt** : PreparedStatement -> oggetto PreparedStatement per eseguire query precompilate.
  - **rs :** ResultSet -> oggetto ResultSet per memorizzare i risultati delle query

  Metodi:

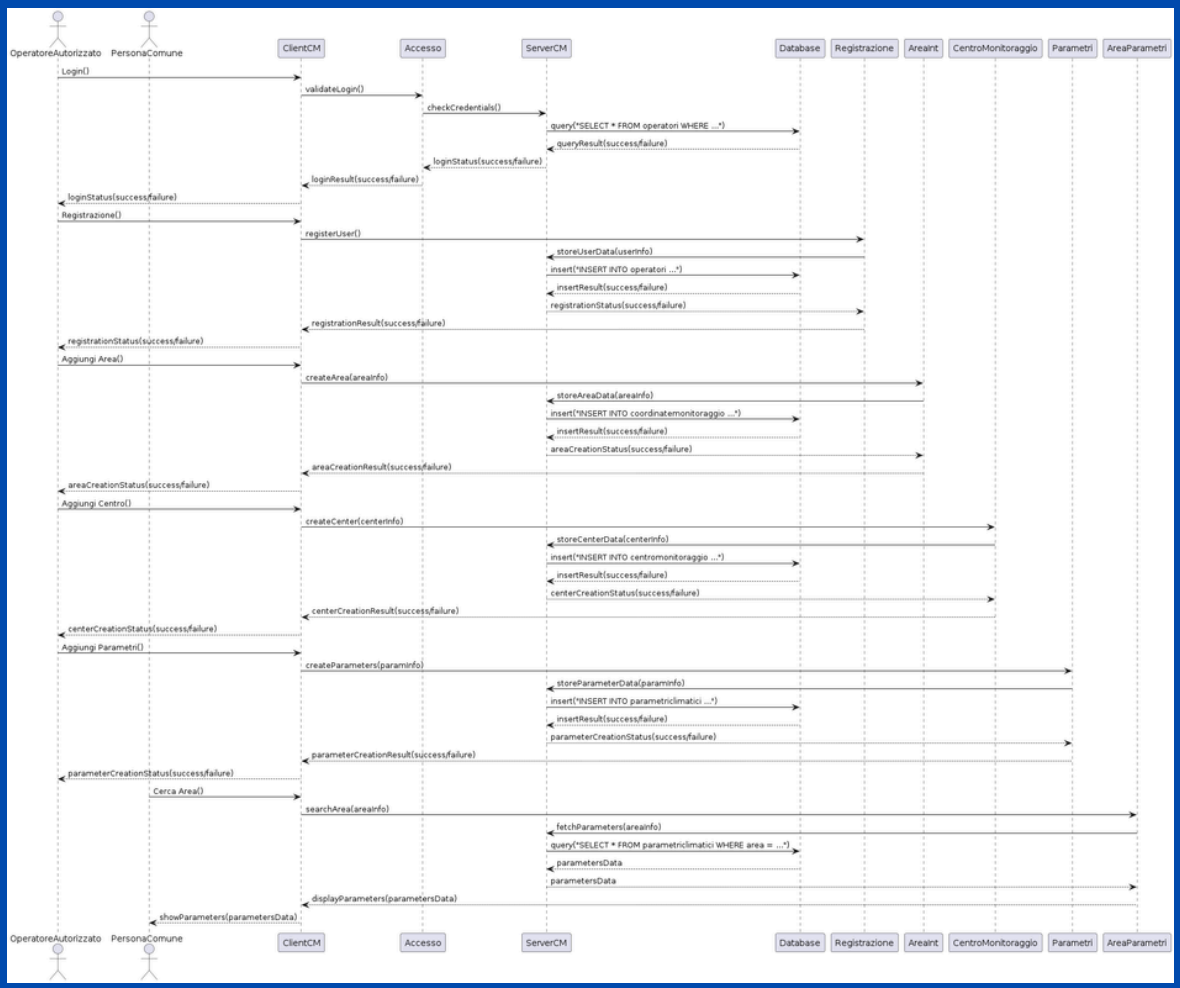
  - Uguali all'interfaccia '**ClimateInterface**'.

Relazioni:

- '**ServerCM**' implementa l'interfaccia '**ClimateInterface**'
- I metodi di 'ServerCM' forniscono l'implementazione concreta dei metodi definiti nell'interfaccia 'ClimateInterface'. Questi metodi includono operazioni come la **connessione e disconnessione dal database**, **l'inserimento e recupero di dati relativi ai centri di montaggio e parametri climatici**, e la **gestione dell'autenticazione** e **registrazione** degli utenti

UML

Il **diagramma di sequenza** rappresenta l'interazione tra gli attori e i componenti del sistema nel tempo, illustrando come i processi funzionano e si comunicano tra loro per portare a termine specifiche funzioni.
Questo diagramma mostra diversi scenari, come il **login**, la r**egistrazione di un utente,** l'**aggiunta di un'area**, l'**aggiunta di un centro** e l'**aggiunta di parametri**, nonché la **ricerca di un'area**.



Attori e componenti:

- **OperatoreAutorizzato** (Attore)
- **PersonaComune** (Attore)
- **ClientCM** (Componenti)
- **Accesso** (Componenti)
- **ServerCM** (Componenti)
- **Database** (Componenti)
- **Registrazione** (Componenti)
- **Area** (Componenti)
- **CentroMontaggio** (Componenti)
- **Parametri** (Componenti)
- **AreaParametri** (Componenti)

UML

Descrizione delle interazioni:

1. **Login**
   - **OperatoreAutorizzato/PersonaComune**: Inviano la richiesta di Logon().
   - **ClientCM**: Chiama validateLogin().
   - **Accesso**: Verifica le credenziali chiamando checkCredential().
   - **ServerCM**: Esegue una query sul database (query(SELECT * FROM operatori WHERE ...)) e restituisce il risultato (queryResult(success/failure)).
   - **Accesso**: Restituisce l'esito del login a ClientCM (loginResult(success/failure)).
   - **ClientCM**: Invia lo stato del login all'attore (loginStatus(success/failure)).
2. **Registrazione**
   - **OperatoreAutorizzato/PersonaComune**: Invia la richiesta di Registrazione().
   - **ClientCM**: Chiama registerUser().
   - **Registrazione**: Memorizza i dati dell'utente (storeUserData(userInfo)) e inserisce i dati nel database (insert(INSERT INTO operatori ...)).
   - **ServerCM**: Conferma l'inserimento (insertResult(success/failure)).
   - **Registrazione**: Restituisce l'esito della registrazione a ClientCM (registrationResult(success/failure)).
   - **ClientCM**: Invia lo stato della registrazione all'attore (registrationStatus(success/failure)).
3. **Aggiungi Area**
   - **OperatoreAutorizzato**: Invia la richiesta di Aggiungi Area().
   - **ClientCM**: Chiama createArea(areaInfo).
   - **Area**: Memorizza i dati dell'area (storeAreaData(areaInfo)) e inserisce i dati nel database (insert(INSERT INTO coordinate ...)).
   - **ServerCM**: Conferma l'inserimento (insertResult(success/failure)).
   - **Area**: Restituisce l'esito della creazione dell'area a ClientCM (areaCreationResult(success/failure)).
   - **ClientCM**: Invia lo stato della creazione dell'area all'attore (areaCreationStatus(success/failure)).
4. **Aggiungi Centro**
   - **OperatoreAutorizzato**: Invia la richiesta di Aggiungi Centro().
   - **ClientCM**: Chiama createCenter(centerInfo).
   - **CentroMontaggio**: Memorizza i dati del centro (storeCenterData(centerInfo)) e inserisce i dati nel database (insert(INSERT INTO centromontaggio ...)).
   - **ServerCM**: Conferma l'inserimento (insertResult(success/failure)).
   - **CentroMontaggio**: Restituisce l'esito della creazione del centro a ClientCM (centerCreationResult(success/failure)).
   - **ClientCM**: Invia lo stato della creazione del centro all'attore (centerCreationStatus(success/failure)).

UML

5. **Aggiungi Parametro**
   - **OperatoreAutorizzato**: Invia la richiesta di Aggiungi Parametro().
   - **ClientCM**: Chiama createParameter(paramInfo).
   - **Parametri**: Memorizza i dati del parametro (storeParameterData(paramInfo)) e inserisce i dati nel database (insert(INSERT INTO parametriclimatici ...)).
   - **ServerCM**: Conferma l'inserimento (insertResult(success/failure)).
   - **Parametri**: Restituisce l'esito della creazione del parametro a ClientCM (parameterCreationResult(success/failure)).
   - **ClientCM**: Invia lo stato della creazione del parametro all'attore (parameterCreationStatus(success/failure)).
6. **Cerca Area**
   - **OperatoreAutorizzato/PersonaComune**: Invia la richiesta di Cerca Area().
   - **ClientCM**: Chiama searchArea(areaInfo).
   - **AreaParametri**: Recupera i parametri dell'area (fetchParameter(areaInfo)) ed esegue una query sul database (query(SELECT * FROM parametriclimatici WHERE area = ...)).
   - **ServerCM**: Restituisce i dati dei parametri (parametersData).
   - **AreaParametri**: Restituisce i dati a ClientCM (parametersData).
   - **ClientCM**: Mostra i parametri all'attore (showParameters(parametersData)).

Questo diagramma è **essenziale per comprendere come le diverse parti del sistema interagiscono e collaborano per completare le operazioni richieste dagli utenti**.

CLIMATE MONITORING

Università degli Studi dell'Insubria – Laurea Triennale in Informatica

Crediti informazioni:

- *Programma: Climate Monitoring*
- *https://docs.oracle.com/javase/8/docs/api/*
- *https://netbeans.apache.org/wiki/index.html*