

# Climate Monitoring

*Manuale Tecnico*

Autori:

Andrea Tettamanti 745387

Luca Mascetti 752951

Versione: 1.1

Data: 07-02-2024

# Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Progettazione</b>	<b>3</b>
<b>3</b>	<b>Struttura Generale dell'Applicazione</b>	<b>3</b>
<b>4</b>	<b>Package shared</b>	<b>4</b>
4.1	Package interfacesRMI . . . . .	4
4.1.1	DataHandlerInterface . . . . .	5
4.1.2	DataQueryInterface . . . . .	5
4.1.3	LogicCenterInterface . . . . .	6
4.1.4	LogicCityInterface . . . . .	6
4.1.5	LogicOperatorInterface . . . . .	7
4.2	Package record . . . . .	7
4.2.1	QueryCondition . . . . .	8
4.2.2	RecordCenter . . . . .	8
4.2.3	RecordCity . . . . .	8
4.2.4	RecordOperator . . . . .	9
4.2.5	RecordWeather . . . . .	9
4.3	Package utils . . . . .	10
4.3.1	Constants . . . . .	11
4.3.2	Functions . . . . .	11
4.3.3	Interfaces . . . . .	12
<b>5</b>	<b>Package server</b>	<b>12</b>
5.1	Server . . . . .	13
5.2	DataBaseManager . . . . .	13
5.3	Package implementationRMI . . . . .	14
5.3.1	DataHandlerImp . . . . .	15
5.3.2	DataQueryImp . . . . .	16
5.3.3	LogicCenterImp . . . . .	17
5.3.4	LogicCityImp . . . . .	18
5.3.5	LogicOperatorImp . . . . .	19
<b>6</b>	<b>Package client</b>	<b>19</b>
6.1	Main . . . . .	19
6.2	Package models . . . . .	20
6.2.1	CurrentOperator . . . . .	20
6.2.2	MainModel . . . . .	21
6.3	Package GUI . . . . .	21
6.3.1	GUI . . . . .	22
6.3.2	Theme . . . . .	23
6.3.3	Widget . . . . .	23
6.3.4	Package layouts . . . . .	24



6.3.5	Package mainElements . . . . .	25
6.3.6	Package panels . . . . .	27
<b>7</b>	<b>Pattern utilizzati</b>	<b>32</b>

## List of Figures

1	Struttura dei package dell'applicazione.	4
2	UML delle classi del package interfacesRMI	5
3	UML delle classi del package record	7
4	UML delle classi Functions e Constants	10
5	UML delle classi Interfaces	11
6	UML del package server	12
7	UML del package implementationRMI	15
8	UML della classe MainModel	21
9	UML del package GUI	22
10	UML della classe Widget	24
11	UML del package layouts	25
12	UML del package mainElements	26
13	UML del package panels	27
14	UML della classe CityVisualizer	29
15	Costruttore della classe CurrentOperator	32
16	Schema del pattern Singleton	33
17	Interfaccia CurrentUserChangeListener	33
18	Metodo notifyCurrentUserChange	34
19	Schema del pattern Observer	34



## 1 Introduzione

Climate Monitoring è un software client-server sviluppato in Java per il Laboratorio A/B del corso in Informatica dell'Università degli Studi dell'Insubria. Il codice sorgente è stato scritto in Java 17 e il software è stato sviluppato con l'ausilio di Apache Maven per la gestione delle dipendenze e la compilazione del progetto. Il software è stato sviluppato per la gestione di centri di monitoraggio, che inviano dati in tempo reale al server, il quale si occupa di memorizzarli e di fornirli ai client che ne fanno richiesta. Per il lato server è stato utilizzato PostgreSQL 42.7.3.jar che permette la connessione ai database PostgreSQL, mentre per il lato client è stato utilizzato JavaSwing per la creazione dell'interfaccia grafica.

## 2 Progettazione

Il software è stato progettato seguendo un approccio client-server, in cui il server si occupa di ricevere i dati dai client e memorizzarli nel database, mentre i client possono richiedere i dati memorizzati al server.

Per la parte client, viene seguita l'architettura CMV (Controller-Model-View), in cui il controller e il view sono uniti nelle classi dell'interfaccia grafica, mentre il model è separato in un package a parte che permette l'uso dei metodi degli oggetti remoti presenti nel server. Questo viene fatto attraverso l'uso della tecnologia java RMI (Remote Method Invocation).

## 3 Struttura Generale dell'Applicazione

Il codice sorgente nel package `src/main/java` è suddiviso in tre Macro-package:

- **client** nel quale sono presenti il package `GUI` che contiene le classi dell'interfaccia grafica, `mainPackage` che contiene la classe punto di ingresso dell'applicazione e `models` che contiene le classi responsabili della connessione remota ai servizi RMI presenti nel modulo Server;
- **server** contiene le classi che implementano i servizi RMI, la classe che si occupa della connessione al database e la principale che fa partire il server;
- **shared** contiene le classi che sono condivise tra il modulo Client e il modulo Server, come ad esempio le interfacce dei servizi RMI, i record dei dati e le classi che implementano funzioni di utilità.



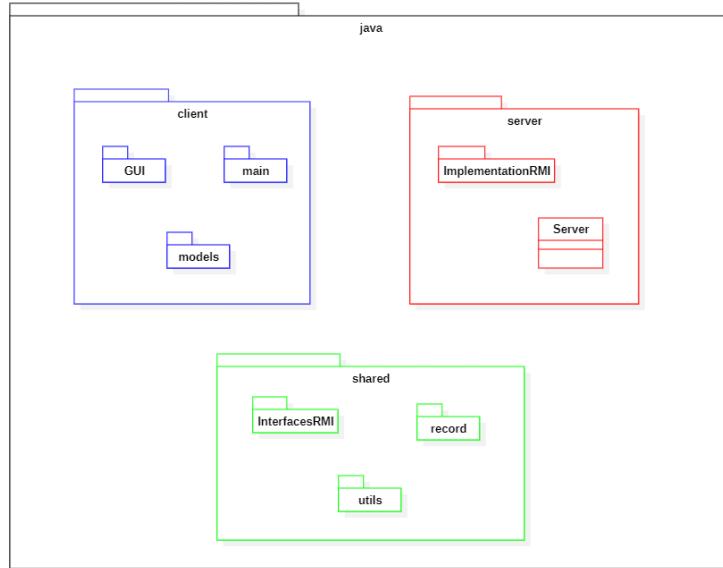


Figure 1: Struttura dei package dell'applicazione.

## 4 Package shared

In questa sezione verranno descritti il package **shared** e le classi che lo compongono.

### 4.1 Package interfacesRMI

In questa sezione verranno descritti il package **interfacesRMI** e le classi che lo compongono.

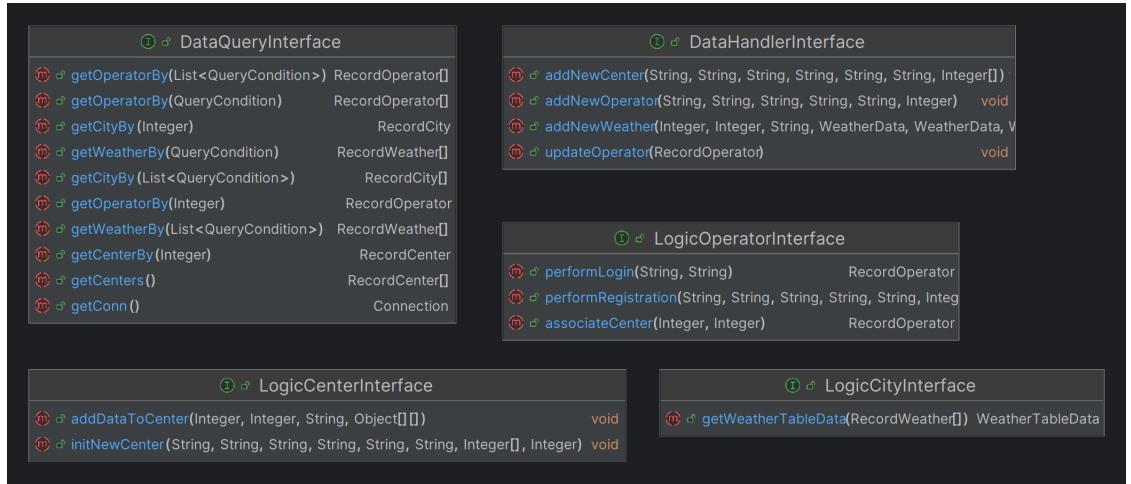


Figure 2: UML delle classi del package interfacesRMI

#### 4.1.1 DataHandlerInterface

L’interfaccia `DataHandlerInterface` è un’interfaccia che permette di definire i metodi remoti che possono essere invocati da un client per interagire con il server per la gestione dei dati. Quest’interfaccia dev’essere implementata dalla classe `DataHandlerImp` la quale esegue i metodi dichiarati. I metodi di tale interfaccia sono:

- `void addNewOperator(String nameSurname, String taxCode, String email, String username, String password, Integer centerID)`
- `RecordCenter addNewCenter(String centerName, String streetName, String streetNumber, String CAP, String townName, String districtName, Integer[] cityIDs)`
- `void addNewWeather(Integer cityID, Integer centerID, String date, RecordWeather.WeatherData wind, RecordWeather.WeatherData humidity, RecordWeather.WeatherData pressure, RecordWeather.WeatherData temperature, RecordWeather.WeatherData precipitation, RecordWeather.WeatherData glacierElevation, RecordWeather.WeatherData glacierMass)`
- `void updateOperator(RecordOperator operator)`

#### 4.1.2 DataQueryInterface

L’interfaccia `DataQueryInterface` è un’interfaccia che permette di definire i metodi remoti che possono essere invocati da un client per interagire con il server per la gestione dei dati. Quest’interfaccia dev’essere implementata dalla classe `DataQueryImp` la quale esegue i metodi dichiarati. I metodi di tale interfaccia sono:



- RecordCity getCityBy(Integer ID)
- RecordCity[] getCityBy(List<QueryCondition> conditions)
- RecordOperator getOperatorBy(Integer ID)
- RecordOperator[] getOperatorBy(QueryCondition condition)
- RecordOperator[] getOperatorBy(List<QueryCondition> conditions)
- RecordCenter getCenterBy(Integer ID)
- RecordCenter[] getCenters()
- RecordWeather[] getWeatherBy(QueryCondition condition)
- RecordWeather[] getWeatherBy(List<QueryCondition> conditions)
- Connection getConn()

#### 4.1.3 LogicCenterInterface

L’interfaccia `LogicCenterInterface` è un’interfaccia che permette di definire i metodi remoti che possono essere invocati da un client per interagire con il server per la gestione dei centri di monitoraggio. Quest’interfaccia dev’essere implementata dalla classe `LogicCenterImp` la quale esegue i metodi dichiarati. I metodi di tale interfaccia sono:

- void initNewCenter(String centerName, String streetName, String streetNumber, String CAP, String townName, String districtName, Integer[] cityIDs, Integer operatorID)
- void addDataToCenter(Integer cityID, Integer operatorID, String date, Object[][] tableData)

#### 4.1.4 LogicCityInterface

L’interfaccia `LogicCityInterface` è utilizzata per esporre i metodi che permettono di ottenere i dati relativi alle città, in particolare i dati metereologici. Quest’interfaccia dev’essere implementata dalla classe `LogicCityImp` la quale esegue i metodi dichiarati. L’unico metodo di tale classe è:

- LogicCityImp.WeatherTableData getWeatherTableData(RecordWeather[] weatherRecords)



#### 4.1.5 LogicOperatorInterface

L’interfaccia `LogicOperatorInterface` è un’interfaccia che permette di definire i metodi remoti che possono essere invocati da un client per interagire con il server per la gestione della logica di business degli operatori. Quest’interfaccia dev’essere implementata dalla classe `LogicOperatorImp` la quale esegue i metodi dichiarati. I metodi di tale interfaccia sono:

- `RecordOperator performLogin(String username, String password)`
- `void performRegistration(String nameSurname, String taxCode, String email, String username, String password, Integer centerID)`
- `RecordOperator associateCenter(Integer operatorID, Integer centerID)`

## 4.2 Package record

In questa sezione verranno descritti il package `record` e le classi che lo compongono.

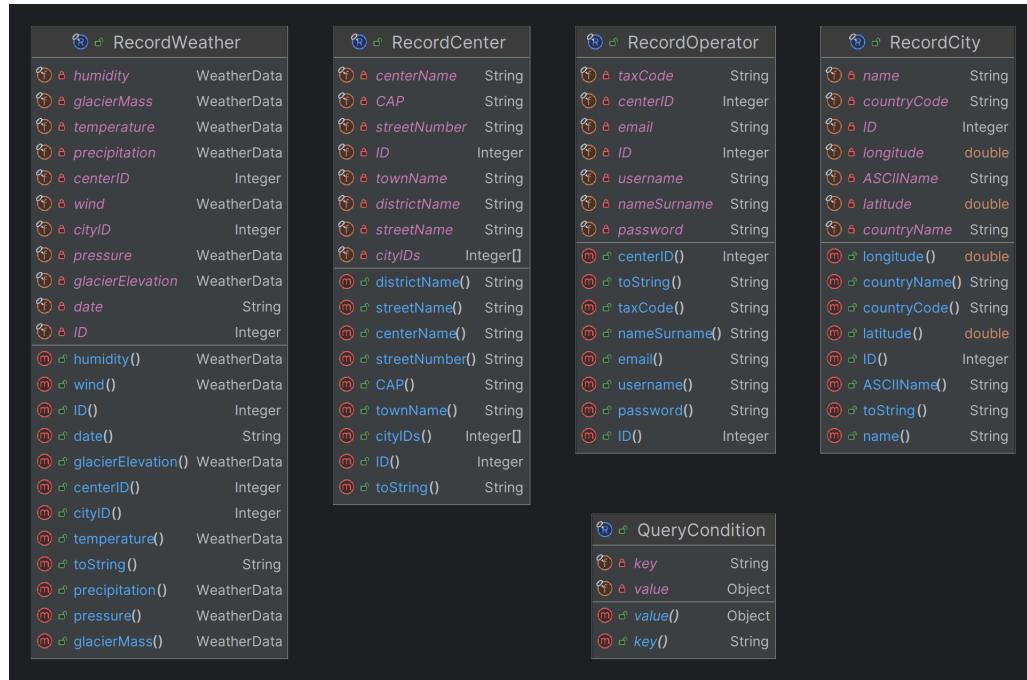


Figure 3: UML delle classi del package `record`



#### 4.2.1 QueryCondition

La classe `QueryCondition` rappresenta una condizione di query.

Viene utilizzata per definire una condizione di ricerca per i record presenti nel database.

#### 4.2.2 RecordCenter

La classe `RecordCenter` rappresenta un centro di monitoraggio e contiene informazioni come il nome del centro, il nome della via, il numero civico, il CAP, il nome del comune, la sigla della provincia e una lista di ID delle città associate. Questa classe è definita come un record, il che significa che è immutabile una volta creata. I metodi di tale classe sono:

- `public String toString()`
- `public Integer ID()`
- `public String centerName()`
- `public String streetName()`
- `public String streetNumber()`
- `public String CAP()`
- `public String townName()`
- `public String districtName()`
- `public String cityIDs()`

#### 4.2.3 RecordCity

La classe `RecordCity` rappresenta una città e contiene informazioni come l'ID della città, il nome, il nome ASCII, il codice del paese, il nome del paese, la latitudine e la longitudine geografica. Questa classe è definita come un record, il che significa che è immutabile una volta creata. I metodi di tale classe sono:

- `public String toString()`
- `public Integer ID()`
- `public String name()`
- `public String ASCIIname()`
- `public String countryCode()`
- `public String countryName()`
- `public double latitude()`
- `public double longitude()`



#### 4.2.4 RecordOperator

La classe `RecordOperator` rappresenta un operatore e contiene informazioni come l'ID, il nome e cognome, il codice fiscale, l'email, il nome utente, la password e l'ID del centro di competenza (se assegnato). Questa classe è definita come un record, il che significa che è immutabile una volta creata. I metodi di tale classe sono:

- `public String toString()`
- `public Integer ID()`
- `public String nameSurname()`
- `public String taxCode()`
- `public String email()`
- `public String username()`
- `public String password()`
- `public Integer centerID()`

#### 4.2.5 RecordWeather

La classe `RecordWeather` rappresenta dati meteorologici registrati in una determinata data per una città e un centro specifici. Questa classe contiene informazioni come l'ID, l'ID della città, l'ID del centro, la data e vari dati meteorologici come il vento, l'umidità, la pressione, la temperatura, la precipitazione, l'elevazione del ghiacciaio e la massa del ghiacciaio. Questa classe è definita come un record, il che significa che è immutabile una volta creata. I metodi di tale classe sono:

- `public String toString()`
- `public Integer ID()`
- `public Integer cityID()`
- `public Integer centerID()`
- `public String date()`
- `public WeatherData wind()`
- `public WeatherData humidity()`
- `public WeatherData pressure()`
- `public WeatherData temperature()`
- `public WeatherData precipitation()`



- public WeatherData glacierElevation()
- public WeatherData glacierMass()
- public Integer score()
- public String comment()

### 4.3 Package utils

In questa sezione verranno descritti il package `utils` e le classi che lo compongono.

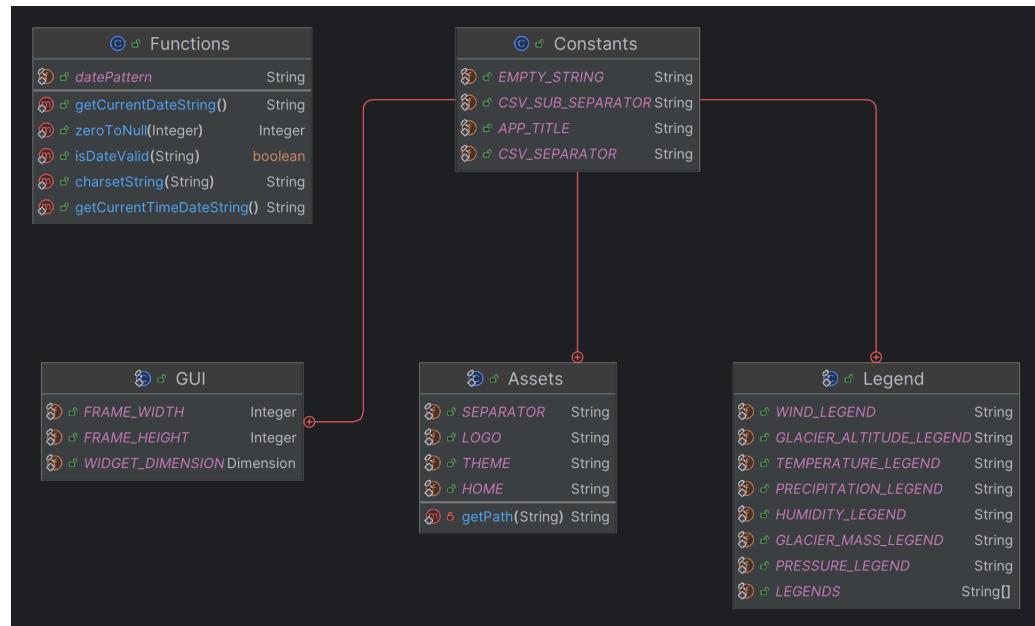


Figure 4: UML delle classi Functions e Constants



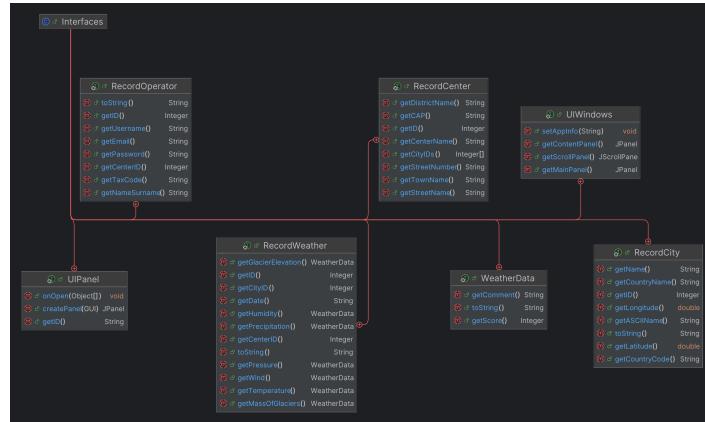


Figure 5: UML delle classi Interfaces

#### 4.3.1 Constants

La classe **Constants** fornisce una serie di costanti e definizioni utilizzate nell'applicazione. Queste costanti includono separatori CSV, titolo dell'applicazione, indici di dati, percorsi dei file, dimensioni GUI predefinite e dati predefiniti. La classe è progettata per memorizzare costanti utilizzate in tutto il codice e semplificare eventuali modifiche future.

#### 4.3.2 Functions

La classe **Functions** fornisce una serie di funzioni di utilità per la gestione delle date e degli orari. Queste funzioni includono la generazione della data corrente, la validazione delle date e la generazione della data e dell'orario correnti. I metodi di tale classe sono:

- **public static String getCurrentDateString():** restituisce una stringa rappresentante la data corrente nel formato **yyyy-MM-dd**.
- **public static boolean isDateValid(String dateString):** verifica se una stringa rappresentante una data è valida e non successiva alla data corrente.
- **public static String getCurrentTimeDateString():** restituisce una stringa rappresentante la data e l'orario correnti nel formato **yyyy-MM-dd HH:mm:ss**.
- **public static String charsetString(String string):** converte una stringa in un formato compatibile con il charset UTF-8.
- **public static Integer zeroToNull(Integer value):** converte un valore 0 in un valore nullo.



### 4.3.3 Interfaces

L’interfaccia **Interfaces** definisce una serie di interfacce e contratti che rappresentano diversi aspetti ed entità all’interno del sistema. Queste interfacce definiscono le proprietà e i metodi che devono essere implementati da classi specifiche per fornire funzionalità legate a città, operatori, centri, dati meteorologici, finestre UI e pannelli UI. Per le interfacce dei record i metodi sono i getter degli attributi. Invece per l’interfaccia di UI Windows i metodi sono:

- JPanel getMainPanel()
  - JScrollPane getScrollPane()
  - JPanel getContentPanePanel()
  - void setAppInfo(String text)

Infine per l'interfaccia di UI Panel i metodi sono:

- JPanel createPanel(GUI gui)
  - void onOpen(Object[] args)
  - String getID()

## 5 Package server

In questa sezione verranno descritti il package **server** e le classi che lo compongono.

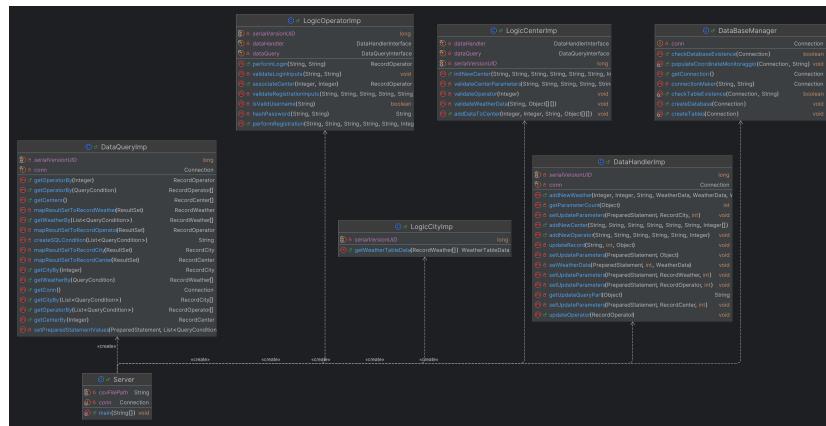


Figure 6: UML del package server



## 5.1 Server

La classe **Server** è progettata per controllare l'esistenza delle tabelle nel database, se non presenti le crea e popola, avviare il registro RMI, pubblicare le implementazioni delle interfacce remote.

Queste implementazioni sono accessibili ai client remoti, che possono invocare metodi per interrogare, gestire e manipolare i dati relativi alle operazioni dell'applicazione. In sintesi, **Server** svolge le seguenti operazioni principali:

- **Creazione connessione al database:** utilizza la classe `DataBaseManager` per stabilire una connessione al database.
- **Creazione e Configurazione delle Tabelle:** utilizza la classe `DataBaseManager` per verificare l'esistenza delle tabelle nel database e, se necessario, crearle.
- **Creazione e Configurazione del Registro RMI:** avvia un registro RMI sulla porta specificata (di default, la porta 1099) utilizzando il metodo `LocateRegistry.createRegistry(int port)`. Questo registro permette ai client remoti di trovare e invocare i metodi delle interfacce registrate.
- **Inizializzazione delle Implementazioni RMI:** la classe crea istanze delle implementazioni delle interfacce remote che gestiscono diverse logiche di business. Queste implementazioni sono
  - `DataQueryImp`
  - `DataHandlerImp`
  - `LogicOperatorImp`
  - `LogicCenterImp`
  - `LogicCityImp`
- **Registrazione delle Implementazioni nel Registro RMI:** ogni implementazione viene registrata nel registro RMI con un nome specifico utilizzando il metodo `rebind(String name, Remote obj)`. Questo passaggio rende i metodi delle interfacce remote accessibili ai client remoti tramite il nome associato.

## 5.2 DataBaseManager

La classe `DataBaseManager` è progettata per gestire le connessioni al database e fornire metodi per creare il database, se non esiste, e le relative tabelle.

Il costruttore `public DataBaseManager(String host, String password)` crea una prima connessione al database `postgres` che, essendo quello creato di default, permette di controllare l'esistenza del database `climatedonitoring` e di crearlo se non esiste.

Poi questa connessione viene chiusa e viene creata una nuova connessione al database `climatedonitoring` con l'utente `postgres` e la password fornita.

I metodi pubblici sono:



- `public boolean checkDatabaseExistence(Connection conn)`: questo metodo controlla se il database `climatemonitoring` esiste già.
- `public static boolean checkTableExistence(Connection conn, String tableName)`: questo metodo controlla se la tabella specificata esiste già nel database.
- `public void createDatabase(Connection conn)`: questo metodo crea il database `climatemonitoring`.
- `public void createTables(Connection conn)`: questo metodo crea le tabelle `coordinatemonitoraggio`, `centrimonitoraggio`, `operatoriregistrati` e `parametriclimatici` concedendo dei permessi specifici per le tabelle create (vedere capitolo riguardante il Database).
- `public static void populateCoordinateMonitoraggio(Connection conn, String csvFilePath)`: questo metodo popola la tabella `coordinatemonitoraggio` con i dati presenti nel file CSV specificato.
- `getConnection()`: questo metodo pubblico restituisce l'oggetto `Connection` gestito da `ConnectionMaker`. Questo permette ad altre classi del sistema di accedere alla connessione per eseguire operazioni di query, aggiornamento o altre interazioni con il database.

I metodi privati sono:

- `connectionMaker(String url, String password)`: questo metodo è responsabile della creazione effettiva della connessione al database. Utilizza l'URL e la password per creare un oggetto `Connection` tramite il `DriverManager` di JDBC. Il metodo configura anche un oggetto `Properties` per gestire le credenziali di accesso.

### 5.3 Package implementationRMI

Il package `implementationRMI` contiene le classi che implementano le interfacce remote definite nel package `interfacesRMI` e che gestiscono le logiche di business dell'applicazione.



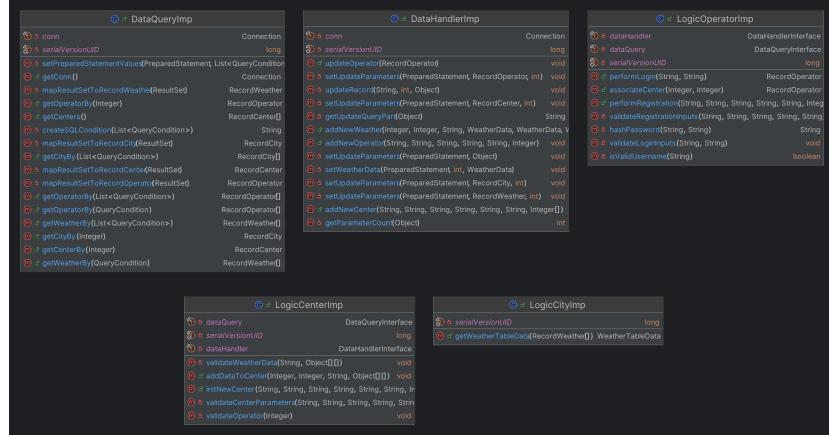


Figure 7: UML del package implementationRMI

### 5.3.1 DataHandlerImp

La classe `DataHandlerImp` implementa l’interfaccia `DataHandlerInterface` e si occupa di gestire operazioni aggiunte e aggiornamento dei record nel database fornendo un’implementazione dei metodi definiti nell’interfaccia.

Questa classe utilizza JDBC per interagire con il database e fornisce metodi per gestire i record relativi agli operatori, ai centri di monitoraggio e ai dati climatici.

I metodi pubblici sono:

- `addNewOperator(String nameSurname, String taxcode, String email, String username, String password, Integer centerID)`: aggiunge un nuovo operatore al database. Prima di inserire i dati, viene effettuato un controllo per assicurarsi che l’utente non esista già. Lancia eccezioni `SQLException`, `RemoteException`, `IllegalArgumentException` in caso di errori.
- `addNewCenter(String centerName, String streetName, String streetNumber, String CAP, String townName, String districtName, Integer[] cityIDs)`: aggiunge un nuovo centro di monitoraggio al database. Prima di inserire i dati, viene effettuato un controllo per evitare duplicati. Restituisce un oggetto `RecordCenter` che rappresenta il centro appena inserito. Lancia eccezioni `SQLException`, `RemoteException`, `IllegalArgumentException` in caso di errori.
- `addNewWeather(Integer cityID, Integer centerID, String date, RecordWeather.WeatherData wind, RecordWeather.WeatherData humidity, RecordWeather.WeatherData pressure, RecordWeather.WeatherData temperature, RecordWeather.WeatherData precipitation, RecordWeather.WeatherData glacierElevation, RecordWeather.WeatherData glacierMass)`: aggiunge nuovi dati climatici al database associati a una



spoeifica città e centro di monitoraggio. Gestisce il formato della data e l'inerimento dei dati climatici in modo dettagliato. Lancia eccezioni `SQLException`, `RemoteException` in caso di errori.

- `updateOperator(RecordOperator operator)`: aggiorna le informazioni di un operatore nel database. Utilizza un metodo interno `updateRecord` per eseguire l'operazione. Lancia eccezioni `SQLException`, `RemoteException` in caso di errori.

I metodi privati sono:

- `updateRecord(String tableName, int ID, Object record)`: esegue l'aggiornamento di un record specifico nel database sulla base del tipo di record e dell'ID fornito.
- `getUpdateQueryPart(Object object)`: genera dinamicamente la stringa SQL necessaria per l'aggiornamento di un record di base al tipo di oggetto (ad esempio, `RecordOperator` o `RecordCenter`).
- `setUpdateParameters(PreparedStatement stmt, Object object)`: imposta i parametri di un oggetto `PreparedStatement` per l'aggiornamento di un record.
- `setWeatherData(PreparedStatement stmt, int index, RecordWeather.WeatherData data)`: imposta i dati climatici (score e commenti) all'interno del `PreparedStatement` per l'inserimento o l'aggiornamento.
- `getParameterCount(Object record)`: restituisce il numero di parametri per un determinante tipo di record, necessario per costruire la query SQL.

### 5.3.2 DataQueryImp

La classe `DataQueryImp` implementa l'interfaccia `DataQueryInterface` e consente di eseguire query sul database per ottenere informazioni relative a città, operatori, centri di monitoraggio e dati climatici fornendo un'implementazione dei metodi definiti nell'interfaccia.

I metodi pubblici sono:

- `getCityBy(Integer ID)`: restituisce un oggetto `RecordCity` contenente le informazioni di una città basandosi sul suo ID.
- `getOperatorBy(Integer ID)`: restituisce un oggetto `RecordOperator` contenente le informazioni di un operatore basandosi sul suo ID.
- `getCenterBy(Integer ID)`: restituisce un oggetto `RecordCenter` contenente le informazioni di un centro di monitoraggio basandosi sul suo ID.
- `getCityBy(List<QueryCondition> conditions)`: restituisce un array di oggetti `RecordCity` contenenti le informazioni di città che soddisfano le condizioni specificate.



- `getOperatorBy(List<QueryCondition> conditions)`: restituisce un array di oggetti `RecordOperator` contenenti le informazioni di operatori che soddisfano le condizioni specificate.
- `getWeatherBy(List<QueryCondition> conditions)`: restituisce un array di oggetti `RecordWeather` contenenti le informazioni di dati climatici che soddisfano le condizioni specificate.
- `getCenters()`: restituisce un array di oggetti `RecordCenter` contenenti le informazioni di tutti i centri di monitoraggio presenti nel database.
- `getConn()`: restituisce l'oggetto `Connection` utilizzato per la connessione al database.

I metodi privati sono:

- `createSQLCondition(List<QueryCondition> conditions)`: crea una stringa SQL di condizione basata su una lista di `QueryCondition`, usata per costruire dinamicamente le query.
- `setPreparedStatementValues(PreparedStatement stmt, List<QueryCondition> conditions)`: imposta i valori di un oggetto `PreparedStatement` basandosi su una lista di `QueryCondition`.
- `mapResultSetToRecordCity ResultSet rs)`: mappa i risultati di una query SQL su un oggetto `RecordCity`.
- `mapResultSetToRecordOperator ResultSet rs)`: mappa i risultati di una query SQL su un oggetto `RecordOperator`.
- `mapResultSetToRecordCenter ResultSet rs)`: mappa i risultati di una query SQL su un oggetto `RecordCenter`.
- `mapResultSetToRecordWeather ResultSet rs)`: mappa i risultati di una query SQL su un oggetto `RecordWeather`.

### 5.3.3 LogicCenterImp

La classe `LogicCenterImp` implementa l'interfaccia `LogicCenterInterface` per la gestione di centri di monitoraggio e dei relativi dati climatici fornendo un'implementazione dei metodi definiti nell'interfaccia.

I metodi pubblici sono:

- `initNewCenter(String centerName, String streetName, String streetNumber, String CAP, String townName, String districtName, Integer[] cityIDs)`: questo metodo crea un nuovo centro di monitoraggio con i dati forniti e associa l'operatore corrente al centro appena creato. Prima di eseguire queste operazioni, il metodo convalida i parametri utilizzando il metodo privato `validateCenterParameters`.



- `addDataToCenter(Integer centerID, Integer operatorID, String date, Object[][] tableDatas)`: questo metodo aggiunge nuovi dati climatici per una città specifica e li associa al centro di monitoraggio gestito dall'operatore indicato. Anche in questo caso, i parametri vengono convalidati prima dell'inserimento.

I metodi privati sono:

- `validateCenterParameters(String centerName, String streetName, String streetNumber, String CAP, String townName, String districtName, Integer[] cityIDs)`: questo metodo privato controlla i parametri forniti per la creazione di un nuovo centro di monitoraggio. Se i parametri non sono validi, viene lanciata un'eccezione.
- `validateOperatorID(Integer operatorID)`: questo metodo privato verifica che l'operatore esista e che sia associato a un centro di monitoraggio.
- `validateWeatherData(String date, Object[][] tableDatas)`: questo metodo privato verifica che la data fornita sia valida e che almeno uno dei dati climatici non sia nullo.

#### 5.3.4 LogicCityImp

La classe `LogicCityImp` implementa l'interfaccia `LogicCityInterface` per la gestione dei dati climatici associati alle città fornendo un'implementazione dei metodi definiti nell'interfaccia.

E' presente la classe interna statica `WeatherTableData` che fornisce i metodi per la gestione dei dati climatici.

I metodi pubblici sono:

- `public Integer getCategoryAvgScore(String category)`: restituisce la media dei punteggi di una categoria specifica.
- `public int getCategoryRecordCount(String category)`: restituisce il numero di record di una categoria specifica.
- `public List<String> getCategoryComments(String category)`: restituisce una lista di commenti relativi a una categoria specifica.
- `public WeatherTableData getWeatherTableData(RecordWeather[] weatherRecords)`: restituisce un oggetto `WeatherTableData` che contiene i dati climatici relativi ai record forniti.

I metodi privati sono:

- `private void processCategory(RecordWeather.WeatherData data, String category)`: questo metodo privato processa i dati climatici di una categoria specifica.



### 5.3.5 LogicOperatorImp

La classe `LogicOperatorImp` implementa l’interfaccia `LogicOperatorInterface` per la gestione degli operatori e delle operazioni di autenticazione fornendo un’implementazione dei metodi definiti nell’interfaccia.

Fornisce metodi per la registrazione di nuovi operatori, il login e le verifiche di validità dei dati delgj operatori.

I metodi pubblici sono:

- `public RecordOperator performLogin(String username, String password)`: esegue il login di un operatore utilizzando le credenziali fornite.
- `public void performRegistration(String nameSurname, String taxcode, String email, String username, String password, Integer centerID)`: registra un nuovo operatore nel database.
- `public RecordOperator associateCenter(Integer operatorID, Integer centerID)`: modifica i dati dell’operatore per associarlo a un centro di monitoraggio specifico.

I metodi privati sono:

- `private boolean isValidUsername(String username)`: verifica che il nome utente fornito sia valido e che non ci sia un duplicato nel database.
- `private String hashPassword(String password)`: genera un hash della password fornita per la memorizzazione sicura nel database.
- `private boolean validateLoginInputs(String username, String password)`: verifica che i dati di login forniti siano validi.
- `private boolean validateRegistrationInputs(String nameSurname, String taxcode, String email, String username, String password)`: verifica che i dati di registrazione forniti siano validi.

## 6 Package client

In questa sezione verranno descritti il package `client` e le classi che lo compongono.

### 6.1 Main

La classe `Main` è il punto di ingresso dell’applicazione client. Essa si occupa di inizializzare il modello principale (`MainModel`) e di lanciare l’interfaccia grafica utente (GUI). Il metodo `main` avvia l’applicazione creando un’istanza di `Main` e chiamando il metodo `launchGUI`.

I metodi di tale classe sono:



- `public void launchGUI()`: fa partire l’interfaccia grafica dell’applicazione.
- `public static void main(String[] args)`: metodo di ingresso dell’applicazione.

## 6.2 Package models

In questa sezione verranno descritti il package `models` e le classi che lo compongono.

### 6.2.1 CurrentOperator

La classe `CurrentOperator` è un singleton che gestisce lo stato dell’operatore attualmente loggato. Fornisce metodi per settare e recuperare l’operatore corrente, controllare lo stato di login, effettuare il logout e gestire i listener che osservano i cambiamenti dell’operatore corrente. Questa classe usa il pattern Singleton per assicurarsi che ci sia una sola istanza di `CurrentOperator`. I metodi di tale classe sono:

- `public static CurrentOperator getInstance()`: restituisce l’istanza corrente di `CurrentOperator`.
- `public void setCurrentOperator(RecordOperator operator)`: imposta l’operatore corrente.
- `public RecordOperator getCurrentOperator()`: restituisce l’operatore attualmente loggato.
- `public boolean isUserLogged()`: controlla se l’operatore corrente è loggato.
- `public void performLogout()`: setta l’operatore corrente a null.
- `void onCurrentUserChange(RecordOperator newOperator)`: metodo chiamato quando l’operatore corrente cambia.
- `public void addCurrentUserChangeListener(CurrentUserChangeListener listener)`: aggiunge un listener per osservare i cambiamenti dell’operatore corrente.
- `public void removeCurrentUserChangeListener(CurrentUserChangeListener listener)`: rimuove un listener dalla lista di osservatori.
- `private void notifyCurrentUserChange()`: notifica tutti i listener registrati che l’operatore corrente è cambiato.



### 6.2.2 MainModel

La classe `MainModel` è responsabile della connessione ai servizi RMI (Remote Method Invocation) e del recupero delle interfacce remote. Queste interfacce permettono al client di comunicare con il server ed eseguire operazioni remote come la gestione dei dati, le query sui dati e la logica operativa. Il costruttore della classe si occupa di localizzare il registro RMI e di effettuare il lookup delle interfacce remote. In caso di errore durante il processo di lookup, viene lanciata una `RuntimeException`.

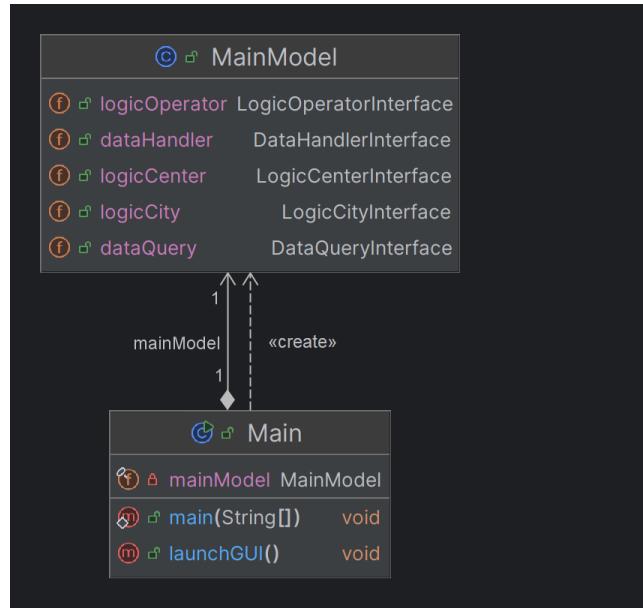


Figure 8: UML della classe MainModel

Oltre agli attributi è presente solo un costruttore:

- `public MainModel():` ottiene il registro RMI creato dal server e cerca le interfacce remote necessarie per la comunicazione con lo stesso.

## 6.3 Package GUI

In questa sezione verranno descritti il package GUI e le classi che lo compongono.



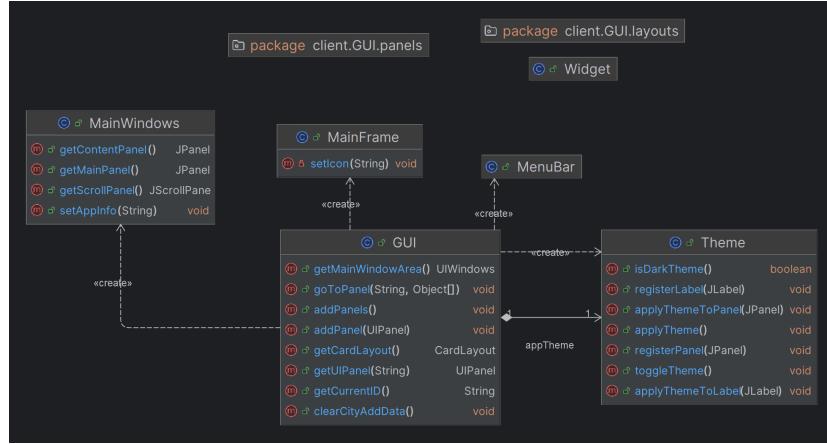


Figure 9: UML del package GUI

### 6.3.1 GUI

La classe `GUI` gestisce l’interfaccia utente dell’applicazione e la navigazione tra diversi pannelli. È una componente chiave dell’architettura dell’applicazione, responsabile della creazione e gestione dei pannelli dell’interfaccia utente e della loro visualizzazione. I metodi di tale classe sono:

- `public void addPanels()`: aggiunge tutti i pannelli utilizzati nell’applicazione.
- `public void addPanel(Interfaces.UIPanel Panel)`: aggiunge un pannello alla mappa dei pannelli e gli applica il tema grafico.
- `public void clearCityAddData()`: pulisce i dati inseriti nel pannello di aggiunta dati di una città.
- `public Interfaces.UIPanel getUIPanel(String ID)`: ottiene un pannello dell’interfaccia utente in base al suo ID.
- `public Interfaces.UIWindows getMainWindowArea()`: restituisce l’area principale dell’applicazione.
- `public CardLayout getCardLayout()`: restituisce il layout a schede utilizzato per la navigazione tra i pannelli.
- `public String getCurrentID()`: restituisce l’ID del pannello corrente.
- `public void goToPanel(String ID, Object[] args)`: cambia il pannello corrente in base all’ID specificato.



### 6.3.2 Theme

La classe `Theme` gestisce il tema grafico dell'applicazione, inclusa la modalità chiaro/scuro, e applica il tema alle etichette (@code `JLabel`) e ai pannelli (@code `JPanel`). La classe consente di passare tra modalità chiara e scura e applica automaticamente il tema corrente a tutti i componenti registrati. I metodi di tale classe sono:

- `public void toggleTheme()`: passa tra modalità chiara e scura.
- `public boolean isDarkTheme()`: controlla se il tema corrente è scuro.
- `public void registerLabel(JLabel label)`: registra un'etichetta per applicare il tema corrente.
- `public void registerPanel(JPanel panel)`: registra un pannello per applicare il tema corrente.
- `public void applyTheme()`: applica il tema corrente a tutti i componenti registrati.
- `public void applyThemeToPanel(JPanel panel)`: applica il tema corrente a un pannello.
- `public void applyThemeToLabel(JLabel label)`: applica il tema corrente a un'etichetta.

### 6.3.3 Widget

La classe `Widget` fornisce componenti grafici comuni utilizzati nell'interfaccia utente dell'applicazione tramite classi interne.

Include pannelli di formattazione, pulsanti con cursori personalizzati, etichette per immagini e oggetti per elementi di una lista a discesa. Questi componenti sono progettati per facilitare la creazione di interfacce utente coerenti e ben stilizzate.



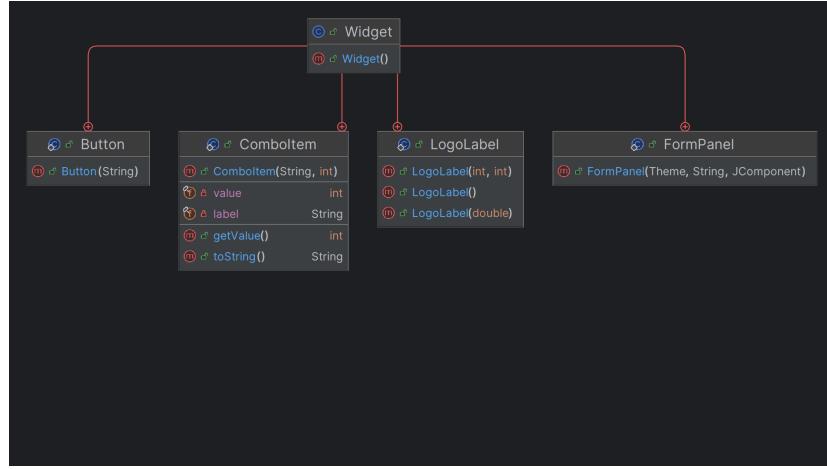


Figure 10: UML della classe Widget

I costruttori e metodi delle classi interne sono:

- `public FromPanel(Theme appTheme, String labelText, JComponent activeArea):` costruttore del pannello di formattazione personalizzato.
- `public Button(String text):` costruttore di un pulsante con il testo specificato.
- `public LogoLabel():` crea un'etichetta con le dimensioni predefinite.
- `public LogoLabel(double scale):` crea un'etichetta con le dimensioni specificate per il logo.
- `public LogoLabel(int width, int height):` crea un'etichetta dimensioni personalizzate.
- `public ComboItem(String label, int value):` costruttore di un oggetto per un elemento di una lista a discesa con un'etichetta e un valore associato.
- `public int getValue():` restituisce il valore associato all'elemento della lista a discesa.
- `public String toString():` restituisce l'etichetta dell'elemento della lista a discesa.

#### 6.3.4 Package layouts

In questo package sono presenti le classi che definiscono i layout utilizzati nell'interfaccia grafica dell'applicazione.



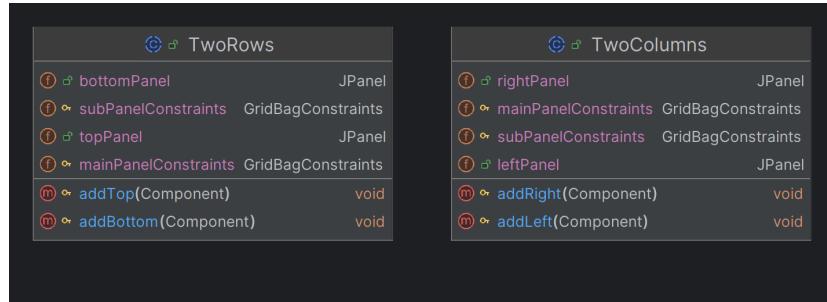


Figure 11: UML del package layouts

**TwoColumns** La classe astratta `TwoColumns` rappresenta un layout a due colonne, con un pannello sinistro e un pannello destro. È progettata per essere estesa da altre classi che necessitano di questo tipo di layout. Entrambi i pannelli utilizzano un `GridBagLayout` per permettere un layout flessibile dei componenti. La classe fornisce metodi protetti per aggiungere componenti ai pannelli sinistro e destro.

I metodi di tale classe sono:

- `protected void addLeft(Component component)`: aggiunge un componente al pannello sinistro.
- `protected void addRight(Component component)`: aggiunge un componente al pannello destro.

**TwoRows** La classe astratta `TwoRows` rappresenta un layout a due righe per un’interfaccia grafica Swing. Le due righe contengono un pannello superiore e un pannello inferiore per organizzare i componenti dell’interfaccia.

I metodi di tale classe sono:

- `protected void addTop(Component component)`: aggiunge un componente al pannello superiore.
- `protected void addBottom(Component component)`: aggiunge un componente al pannello inferiore.

### 6.3.5 Package mainElements

In questo package sono presenti le classi che definiscono gli elementi della finestra principale dell’applicazione.



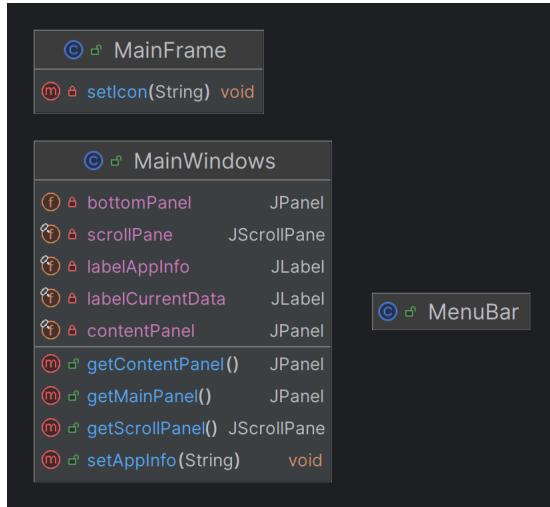


Figure 12: UML del package mainElements

**MainFrame** La classe **MainFrame** rappresenta il frame principale dell'applicazione. Il frame contiene i componenti principali dell'interfaccia utente e funge da contenitore principale per tutti i widget e pannelli dell'applicazione. I costruttori e metodi di tale classe sono:

- `public MainFrame():` configura il frame principale dell'applicazione.
- `private void setIcon(String iconPath):` imposta l'icona del frame.

**MainWindows** La classe **MainWindows** rappresenta la finestra principale dell'applicazione. Questa finestra contiene un pannello scorrevole con un layout a schede, in cui vengono visualizzate diverse schermate dell'applicazione. Inoltre, nella parte inferiore della finestra, vengono visualizzate informazioni sull'operatore attualmente loggato e l'orario corrente. I costruttori e metodi di tale classe sono:

- `public MainWindows(CardLayout cardLayout):` inizializza la finestra principale dell'applicazione impostando il pannello a scorrimento, il layout a schede e aggiungendo timer e informazioni sull'operatore nella parte inferiore della finestra.
- `public JPanel getMainPanel():` restituisce il pannello principale della finestra.
- `public JScrollPane getScrollPane():` restituisce il pannello scorrevole della finestra.
- `public JPanel getContentPane():` restituisce il pannello contenente i pannelli a schede.



- `public void setAppInfo(String text)`: imposta le informazioni sull'operatore attualmente loggato.

**MenuBar** La classe `MenuBar` rappresenta la barra del menù dell'interfaccia grafica dell'applicazione. Questa barra del menù consente la navigazione tra diverse sezioni dell'applicazione e fornisce opzioni per cambiare il tema dell'interfaccia utente e gestire la sessione dell'operatore. Gli elementi principali del menù includono home, ricerca città, e un sotto-menù per l'area operatore con le opzioni di login, registrazione, gestione città e logout. All'interno è presente solo il costruttore:

- `public MenuBar(GUI gui)`: imposta il layout della barra del menu, aggiunge gli elementi del menu e gestisce gli eventi di selezione del menu.

### 6.3.6 Package panels

In questo package sono presenti le classi che definiscono i pannelli dell'interfaccia grafica dell'applicazione.

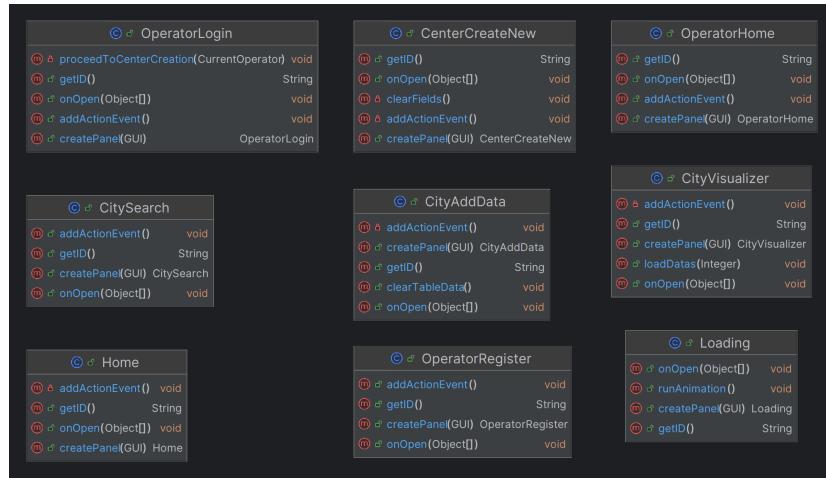


Figure 13: UML del package panels

**Loading** La classe `Loading` rappresenta un pannello di caricamento animato che viene visualizzato all'avvio dell'applicazione. Questo pannello mostra il nome dell'applicazione con una serie di punti che si muovono per simulare un caricamento. L'animazione prosegue fino a quando il pannello non reindirizza automaticamente all'homepage dell'applicazione. I metodi di tale classe sono:

- `public void runAnimation()`: avvia l'animazione di caricamento.
- `public Loading createPanel(GUI gui)`: crea il pannello di caricamento.



- `public String getID()`: restituisce l'ID del pannello.
- `public void onOpen(Object[] args)`: gestisce l'apertura del pannello.

**Home** La classe Home rappresenta il pannello principale dell'applicazione visualizzato dopo il caricamento iniziale. Questo pannello fornisce all'utente due opzioni principali: **Cerca e visualizza dati** per accedere alla funzionalità di ricerca e visualizzazione dei dati, e **Gestisci area operatore** per accedere alla gestione dell'area riservata agli operatori. L'utente può selezionare una delle opzioni per avviare le funzionalità specifiche dell'applicazione. I metodi di tale classe sono:

- `private void addActionEvent()`: aggiunge gli eventi per la navigazione tra i pannelli.
- `public Home createPanel(GUI gui)`: crea il pannello home.
- `public String getID()`: restituisce l'ID del pannello.
- `public void onOpen(Object[] args)`: gestisce l'apertura del pannello.

**CitySearch** La classe CitySearch rappresenta il pannello per effettuare ricerche sulla base di dati delle città. Gli utenti possono cercare una città per nome o per coordinate geografiche utilizzando i campi di input e i pulsanti forniti. I metodi di tale classe sono:

- `public void addActionEvent()`: aggiunge gli eventi per la ricerca delle città in base al pulsante premuto.
- `public CitySerch createPanel(GUI gui)`: crea il pannello di ricerca città.
- `public String getID()`: restituisce l'ID del pannello.
- `public void onOpen(Object[] args)`: gestisce l'apertura del pannello.

**CityVisualizer** La classe CityVisualizer rappresenta il pannello per la visualizzazione dei dati di una città, inclusi i dati meteorologici relativi a diverse categorie. È utilizzato nell'applicazione per mostrare dettagli sulla città selezionata e i dati meteorologici associati. I metodi di tale classe sono:

- `private void addActionEvent()`: aggiunge gli eventi al pulsante per tornare alla schermata precedente.
- `public void loadDatas(Integer cityID)`: carica i dati della città selezionata.
- `public CityVisualizer createPanel(GUI gui)`: crea il pannello di visualizzazione dei dati della città.



- `public String getID()`: restituisce l'ID del pannello.
- `public void onOpen(Object[] args)`: gestisce l'apertura del pannello.

Sono presenti anche due classi interne:

- `TooltipCellRenderer`: classe interna che gestisce il rendering delle celle della tabella.
- `NonEditableCellEditor`: classe interna per l'editor delle celle non modificabili.

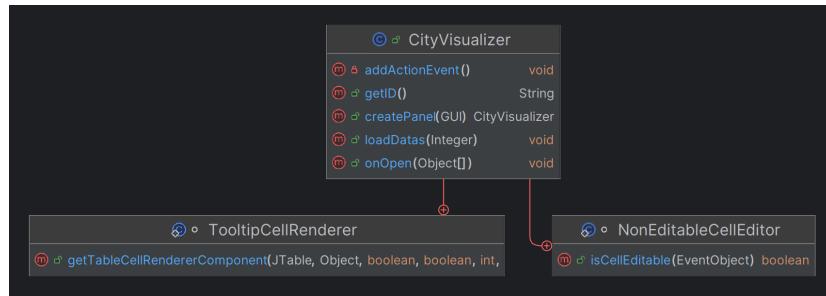


Figure 14: UML della classe CityVisualizer

I metodi di tali classi interne sono:

- `public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column)`: restituisce il componente per il rendering delle celle della tabella.
- `public boolean isCellEditable(EventObject e)`: restituisce `false` per indicare che la cella non è modificabile.

**OperatorHome** La classe `OperatorHome` rappresenta il pannello principale per gli operatori dell'applicazione. Da questo pannello, gli operatori possono scegliere di registrarsi o accedere all'applicazione. Questa classe gestisce la navigazione tra il pannello di registrazione e quello di login tramite i pulsanti corrispondenti. I metodi di tale classe sono:

- `public void addActionEvent()`: aggiunge gli eventi per la navigazione tra i pannelli in base al pulsante premuto.
- `public String getID()`: restituisce l'ID del pannello.
- `public void onOpen(Object[] args)`: gestisce l'apertura del pannello.

**OperatorRegister** La classe `OperatorRegister` rappresenta il pannello utilizzato per la registrazione di un operatore all'interno dell'applicazione. Questo pannello consente agli operatori di inserire i loro dati personali, come nome, codice fiscale, email, username e password, al fine di creare un nuovo account operatore. Utilizza il modulo server RMI per la registrazione e gestisce le eccezioni che possono derivare dalla connessione al server o dalle operazioni sul database. I metodi di tale classe sono:

- `public void addActionEvent()`: aggiunge gli eventi per la registrazione dell'operatore.
- `public OperatorRegister createPanel(GUI gui)`: crea il pannello di registrazione operatore.
- `public String getID()`: restituisce l'ID del pannello.
- `public void onOpen(Object[] args)`: gestisce l'apertura del pannello.

**OperatorLogin** La classe `OperatorLogin` rappresenta il pannello di login per gli operatori dell'applicazione. Gli operatori possono inserire il loro username e la password per accedere all'applicazione. Utilizza un modulo server RMI per autenticare l'operatore e interagisce con un database per recuperare e gestire i dati necessari. I metodi di tale classe sono:

- `public void addActionEvent()`: aggiunge gli eventi per il login dell'operatore.
- `public OperatorLogin createPanel(GUI gui)`: crea il pannello di login operatore.
- `public String getID()`: restituisce l'ID del pannello.
- `public void onOpen(Object[] args)`: gestisce l'apertura del pannello.
- `private void proceedToCenterCreation(CurrentOperator currentOperator)`: metodo privato che gestisce la creazione o associazione di un centro per l'operatore che ha effettuato il login per la prima volta.

**CenterCreateNew** La classe `CenterCreateNew` rappresenta il pannello per la creazione di un nuovo centro di monitoraggio da parte dell'operatore. Il pannello consente all'operatore di inserire informazioni sul centro, come il nome, la via, il numero civico, il CAP, il comune, la provincia e le città associate al centro.

Una volta inseriti i dati, l'operatore può salvare il centro nel sistema utilizzando i servizi offerti dal modulo server RMI e interagendo con il database.

La classe gestisce la validazione dei dati inseriti e fornisce feedback all'operatore in caso di errori.

La comunicazione con il server RMI è gestita attraverso l'interfaccia `DataQueryImp` per le query sui dati e `MainModel` per la logica di applicazione. I metodi di tale classe sono:



- `private void clearFields()`: pulisce i campi di input del pannello.
- `private void addActionEvent()`: aggiunge gli eventi per la creazione di un nuovo centro.
- `public CenterCreateNew createPanel(GUI gui)`: crea il pannello di creazione di un nuovo centro.
- `public String getID()`: restituisce l'ID del pannello.
- `public void onOpen(Object[] args)`: gestisce l'apertura del pannello.

**CityAddData** La classe `CityAddData` rappresenta il pannello per l'aggiunta di dati di una città da parte dell'operatore.

Il pannello consente all'operatore di inserire i dati relativi alla città selezionata quali: data di rilevamento dati, punteggi per le varie categorie ed eventualmente dei commenti che li descrivono.

La classe gestisce la validazione della data inserita, il limite di caratteri per i commenti dei dati e la funzionalità per il salvataggio dei dati inseriti. I metodi di tale classe sono:

- `public void clearTableData()`: pulisce i dati inseriti nella tabella.
- `private void addActionEvent()`: aggiunge gli eventi per l'aggiunta dei dati della città.
- `public CityAddData createPanel(GUI gui)`: crea il pannello di aggiunta dati di una città.
- `public String getID()`: restituisce l'ID del pannello.
- `public void onOpen(Object[] args)`: gestisce l'apertura del pannello.

Anche in questo caso sono presenti delle classi interne:

- `IntegerCellEditor`: classe interna per l'editor delle celle numeriche.
- `TooltipCellRenderer`: classe interna che gestisce il rendering delle celle della tabella.
- `NonEditableCellEditor`: classe interna per l'editor delle celle non modificabili.

I metodi di tali classi interne sono:

- `public Integer getCellEditorValue()`: restituisce il valore intero selezionato dall'utente.
- `public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column)`: restituisce il componente per il rendering delle celle della tabella.
- `public boolean isCellEditable(EventObject e)`: restituisce `false` per indicare che la cella non è modificabile.



## 7 Pattern utilizzati

All'interno della classe `CurrentOperatore.java` sono stati utilizzati due pattern specifici: il **Singleton** e l'**Observer**.

Il pattern Singleton è utilizzato per garantire che ci sia una sola istanza della classe `CurrentOperator` nell'applicazione. La classe ha un costruttore privato e un campo statico `instance` che rappresenta l'istanza unica della classe. Il metodo `getInstance()` restituisce l'istanza esistente se è già stata creata o ne crea una nuova se non esiste ancora.

```
public class CurrentOperator {
    private static CurrentOperator instance = null;
    private RecordOperator currentOperator = null;
    private List<CurrentUserChangeListener> listeners = new ArrayList<>();

    // Costruttore privato per implementare il pattern Singleton
    private CurrentOperator() {
    }

    /**
     * ...
     */
    public static CurrentOperator getInstance() {
        if (instance == null) {
            instance = new CurrentOperator();
        }
        return instance;
    }
}
```

Figure 15: Costruttore della classe `CurrentOperator`

Questo pattern è stato implementato seguendo questo schema:



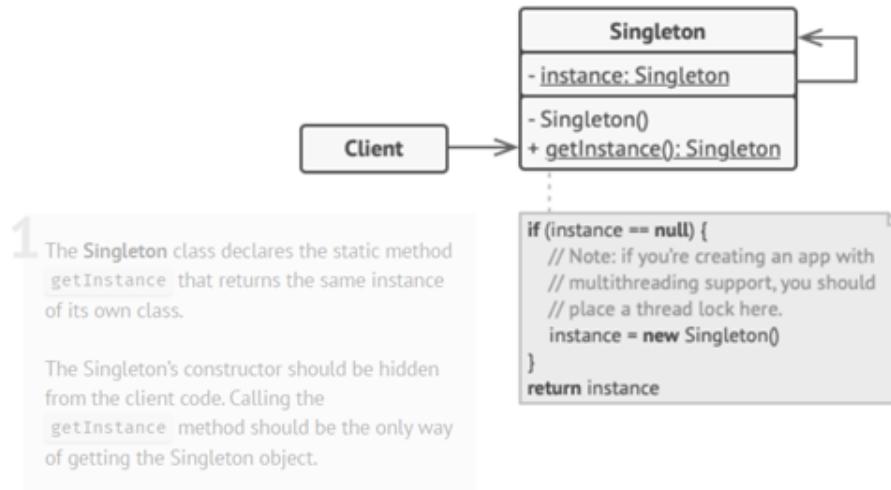


Figure 16: Schema del pattern Singleton

Il pattern Observer è utilizzato per notificare altri oggetti quando l'utente corrente cambia. La classe `CurrentUserChangeListener` definisce un'interfaccia `CurrentUserChangeListener` che deve essere implementata da tutte le classi interessate ai cambiamenti dell'utente corrente.

```

public interface CurrentUserChangeListener {
    /**
     * ...
     void onCurrentUserChange(RecordOperator newOperator);
}

```

Figure 17: Interfaccia CurrentUserChangeListener

La classe contiene metodi per aggiungere (`addCurrentUserChangeListener`) e rimuovere (`removeCurrentUserChangeListener`) listener interessati ai cambiamenti dell'utente corrente. Quando l'utente corrente cambia, il metodo `notifyCurrentUserChange` viene chiamato per notificare tutti i listener registrati.



```

private void notifyCurrentUserChange() {
    for (CurrentUserChangeListener listener : listeners) {
        listener.onCurrentUserChange(currentOperator);
    }
}

```

Figure 18: Metodo notifyCurrentUserChange

In questo modo, altre parti dell'applicazione possono essere avvise quando l'utente corrente cambia, consentendo una gestione flessibile degli eventi correlati all'utente.

Questo pattern è stato implementato seguendo questo schema:

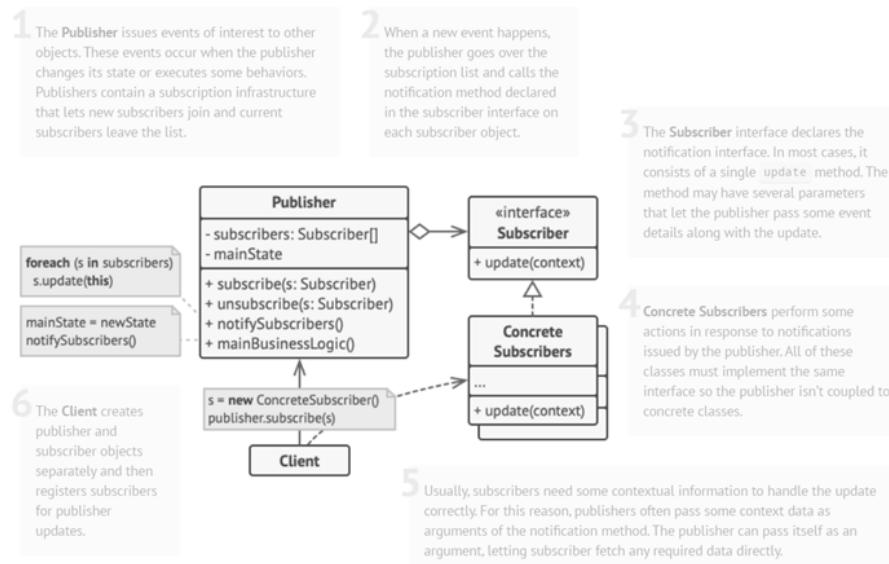


Figure 19: Schema del pattern Observer

