

Climate Monitoring

Manuale Tecnico

Autori:

Andrea Tettamanti 745387

Luca Mascetti 752951

Versione: 1.1

Data: 07-02-2024

Contents

1	Introduzione	3
2	Progettazione	3
3	Struttura Generale dell'Applicazione	3
4	Package server	4
4.1	Server	5
4.2	DataBaseManager	5
4.3	Package implementationRMI	6
4.3.1	DataHandlerImp	6
4.3.2	DataQueryImp	8
4.3.3	LogicCenterImp	9
4.3.4	LogicCityImp	9
4.3.5	LogicOperatorImp	10
5	Package shared	11
5.1	Package interfacesRMI	11
5.1.1	DataHandlerInterface	11
5.1.2	DataQueryInterface	12
5.1.3	LogicCenterInterface	12
5.1.4	LogicCityInterface	13
5.1.5	LogicOperatorInterface	13
5.2	Package record	13
5.2.1	QueryCondition	14
5.2.2	RecordCenter	14
5.2.3	RecordCity	15
5.2.4	RecordOperator	15
5.2.5	RecordWeather	16
5.3	Package utils	16
5.3.1	Constants	17
5.3.2	Functions	18
5.3.3	Interfaces	18
6	Pattern utilizzati	19

List of Figures

1	Struttura dei package dell'applicazione.	4
2	UML del package server	4
3	UML delle classi del package interfacesRMI	11
4	UML delle classi del package record	14
5	UML delle classi Functions e Constants	17
6	UML delle classi Interfaces	17



7	Costruttore della classe CurrentOperator	19
8	Schema del pattern Singleton	20
9	Interfaccia CurrentUserChangeListener	20
10	Metodo notifyCurrentUserChange	21
11	Schema del pattern Observer	21

Climate Monitoring

Manuale Tecnico

Autori:

Andrea Tettamanti 745387

Luca Mascetti 752951

2

Versione: 1.1

Data: 07-02-2024

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.



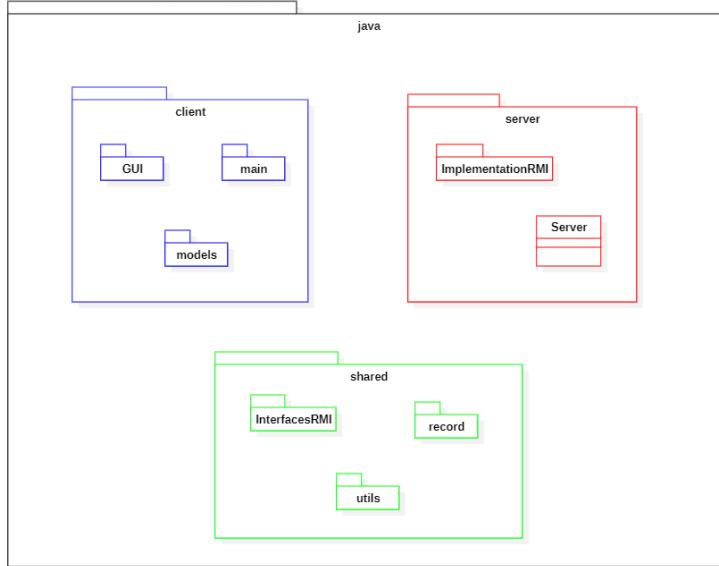


Figure 1: Struttura dei package dell'applicazione.

4 Package server

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

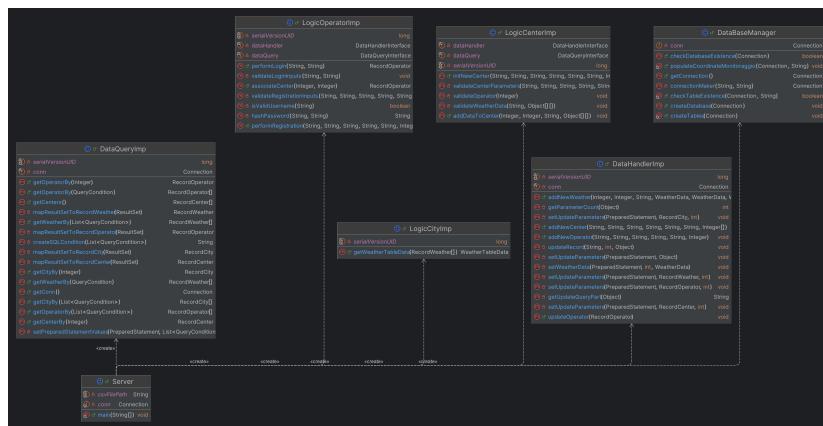


Figure 2: UML del package server

4.1 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.

4.2 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.

4.3 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.

4.3.1 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 specifica città e centro di monitoraggio. Gestisce il formato della data e l'inserimento 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 determinato tipo di record, necessario per costruire la query SQL.



4.3.2 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`.

4.3.3 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.

4.3.4 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.

4.3.5 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.

5 Package shared

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

5.1 Package interfacesRMI

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

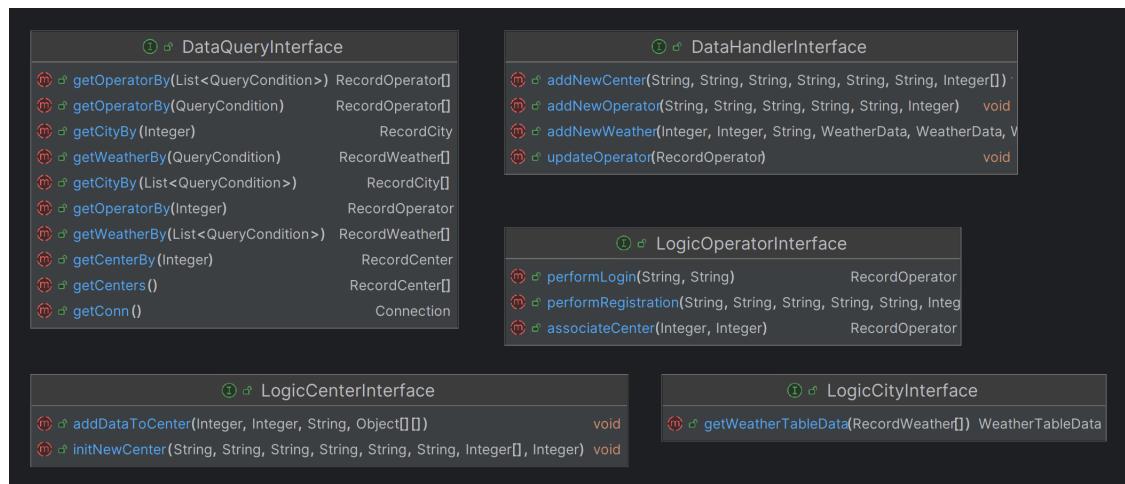


Figure 3: UML delle classi del package `interfacesRMI`

5.1.1 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)`

5.1.2 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()`

5.1.3 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)`



5.1.4 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)`

5.1.5 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)`

5.2 Package record

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



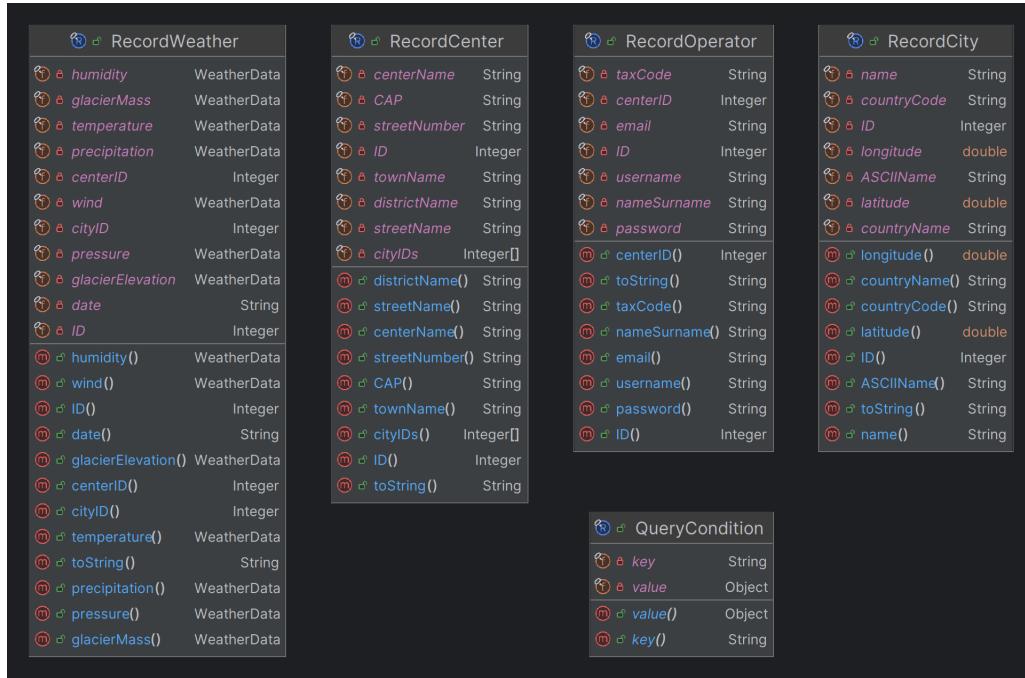


Figure 4: UML delle classi del package record

5.2.1 QueryCondition

La classe `QueryCondition` rappresenta una condizione di query.

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

5.2.2 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()

5.2.3 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()

5.2.4 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()



5.2.5 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()`

5.3 Package utils

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



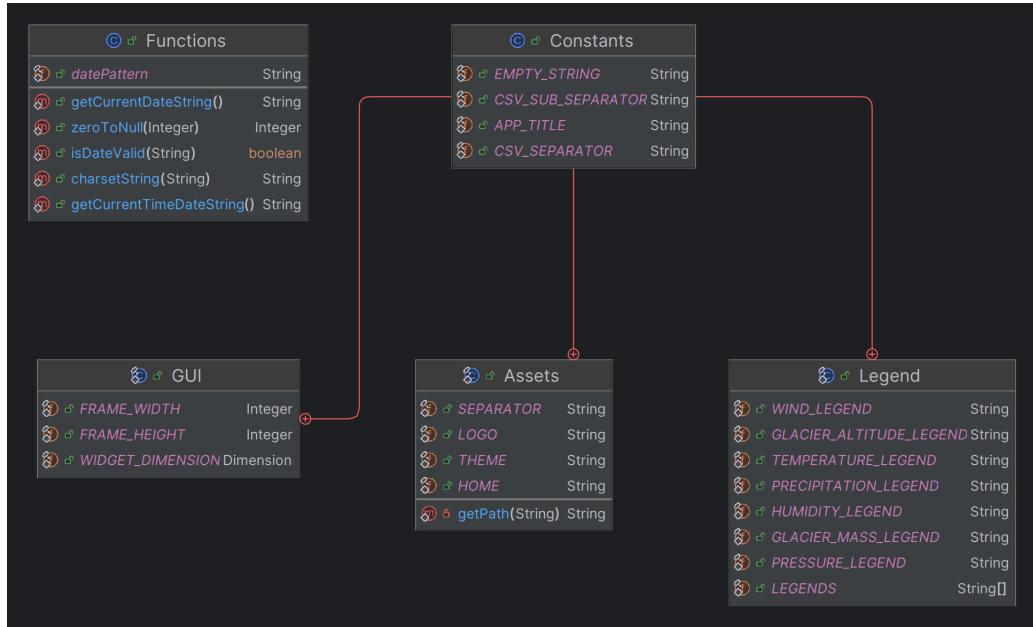


Figure 5: UML delle classi Functions e Constants

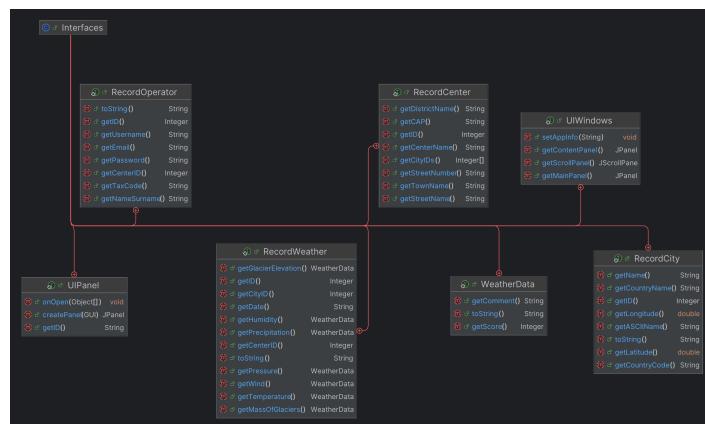


Figure 6: UML delle classi Interfaces

5.3.1 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.

5.3.2 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 isValid(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.

5.3.3 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 getContentPane()`
- `void setAppInfo(String text)`

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

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



6 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 7: Costruttore della classe `CurrentOperator`

Questo pattern è stato implementato seguendo questo schema:



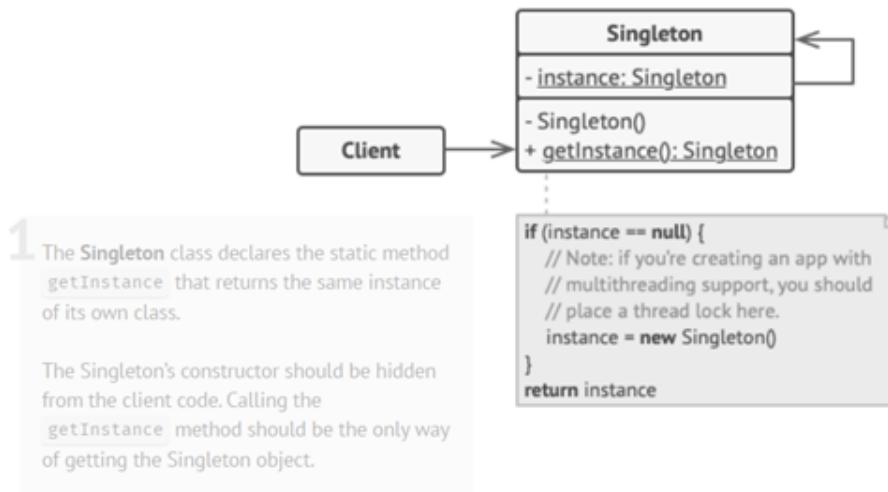


Figure 8: Schema del pattern Singleton

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 9: 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 10: 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:

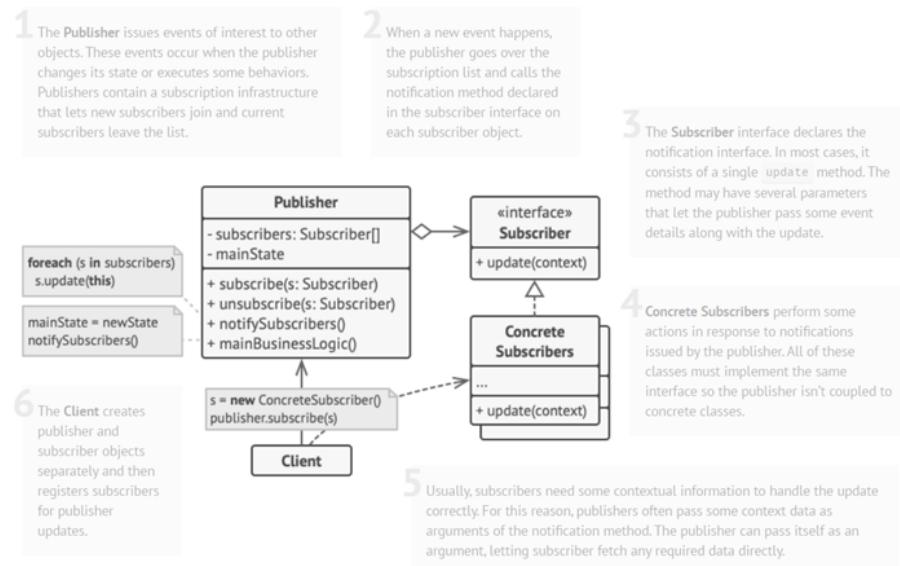


Figure 11: Schema del pattern Observer

