Climate Monitoring

Manuale Tecnico

Autori:

Andrea Tettamanti 745387

Luca Mascetti 752951

Versione: 1.1

Data: 07-02-2024

Contents

List of Figures

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.

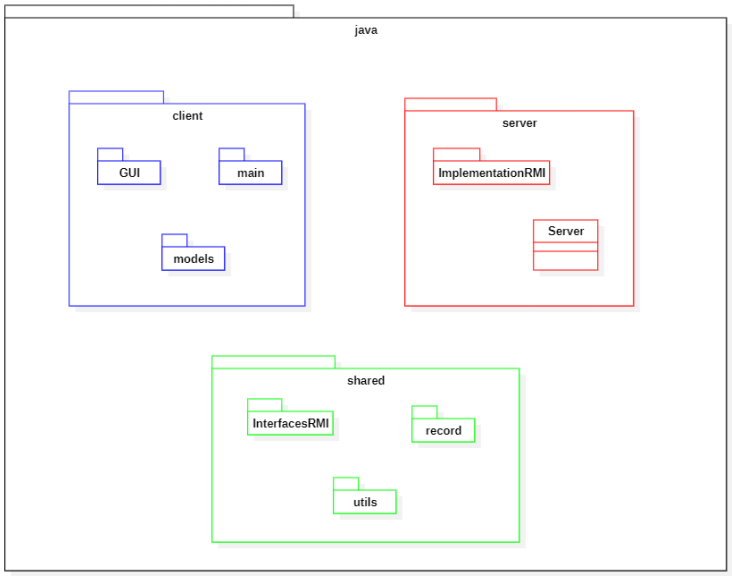

Figure 1: Struttura dei package dell'applicazione.

4 Package Server

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

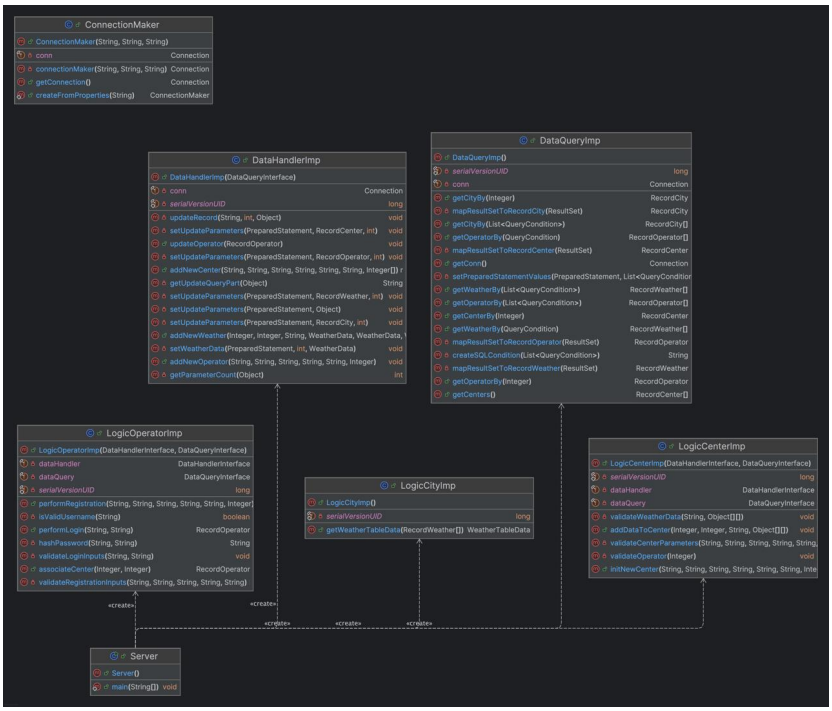

Figure 2: UML del package Server

4.1 Server

La classe `Server` è progettata per avviare il registro RMI e pubblicare le implementazioni delle interfacce remote.
Queste implementazioni sono accessibili ai client remoti, che possoo invocare metodi per interrogare, gestire e manipolare i dati relativi alle operazioni dell'applicazione.
In sisntesi, `Server` svolge le seguenti operazioni principali:

- **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 trmite il nome associato.

4.2 ConnectionMaker

La classe `ConnectionMaker` è progettata per gestire la connessione al database, utilizzando sia parametr5i espliciti che file di configurazione.
La gestione della connessione viene effettuata tramite l'oggetto `Connection`, che rappresenta un collegamento la database, permettendo l'esecuzione di query e altre operazioni di manipolazione dei dati.

I metodi pubblici sono:

- `createFromProperties(String filepath)`: questo metodo statico consente di creare un'istanza di `ConnectionMaker` utilizzando un file di properties. Il file specificato deve contenere le chiavi `db.url`, `db.username` e `db.password`, che vengono lette e utilizzate per stabilire la connessione al databse. Se il file non viene trovato o si verifica un errore durante la lettura, viene lanciata un'eccezione.

- `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 username, String password)`: questo metodo privato è responsabile della creazione effettiva della connessione al databse. Utilizza l'URL, il nome utene e la password per creare un oggetto `Connection` tramite il `DriverManager` di JDBC. Il metodo configura anche un oggetto `Properties` pre 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.

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 databse. 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 districName, 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 spoecifica 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 determianto 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 monitoraggioe e dati climatici.

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.

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 ilo 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.
- `validateOpratorID(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 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.

```java
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 3: Costruttore della classe CurrentOperator

Questo pattern è stato implementato seguendo questo schema:

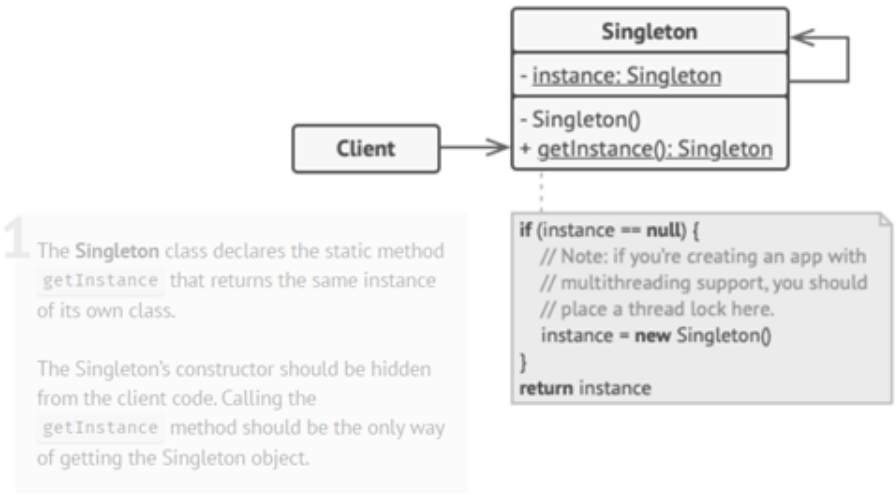

Figure 4: Schema del pattern Singleton

Il pattern Observer è utilizzato per notificare altri oggetti quando l'utente corrente cambia. La classe`CurrentOperator` 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 5: 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 6: Metodo notifyCurrentUserChange

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

Questo pattern è stato implementato seguendo questo schema:

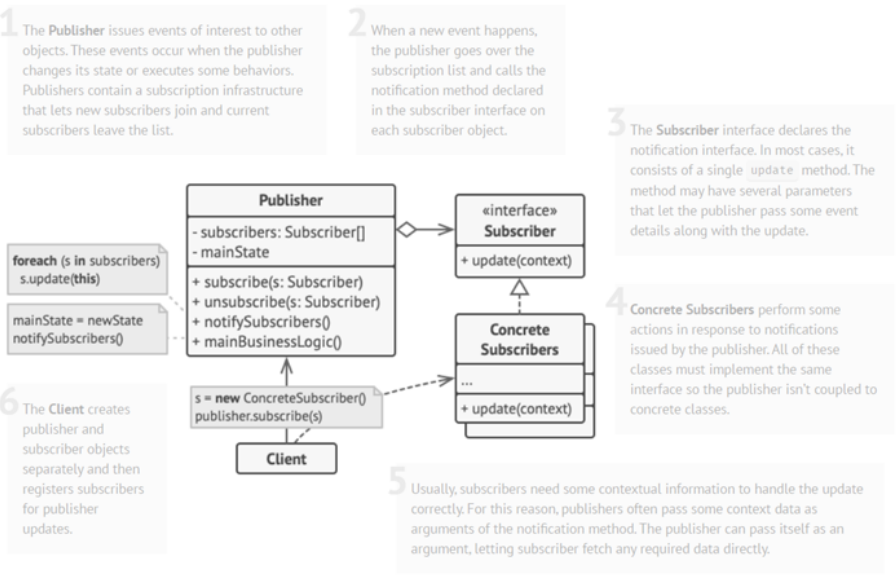

Figure 7: Schema del pattern Observer