Climate Monitoring

Manuale Tecnico

Autori:

Andrea Tettamanti 745387

Luca Mascetti 752951

Versione: 1.1

Data: 07-02-2024

Contents

1 Introduzione **2**

2 Struttura Generale dell'Applicazione **2**

3 Strutture Dati e scelte Algoritmiche **2**

4 Pattern utilizzati **7**

List of Figures

1 Struttura dei package dell'applicazione. 2
2 Costruttore della classe CurrentOperator 7
3 Schema del pattern Singleton 8
4 Interfaccia CurrentUserChangeListener 8
5 Metodo notifyCurrentUserChange 9
6 Schema del pattern Observer 9

1 Introduzione

Climate Monitoring è un progetto sviluppato in Java per il Laboratorio A del corso in Informatica dell'Università degli Studi dell'Insubria. Il progetto è stato sviluppato in Java 17 su OS Windows 10.

2 Struttura Generale dell'Applicazione

L'applicazione è stata sviluppata seguendo l'architettura `MVC (Model-View-Controller)`, dove le parti View e Controller sono inglobati nella User Interface (UI). Di conseguenza il codice sorgente del package `src` è suddiviso in due Macro-package: `Models` in cui sono presenti tutte le classi che gestiscono i dati, mentre nel package `GUI` sono presenti le classi che gestiscono la UI e l'interazione tra i comandi fatti dall'utente e lo storage dei dati. È presente anche un package, `utils`, che contiene classi di utilità usate nella maggior parte delle altre classi.

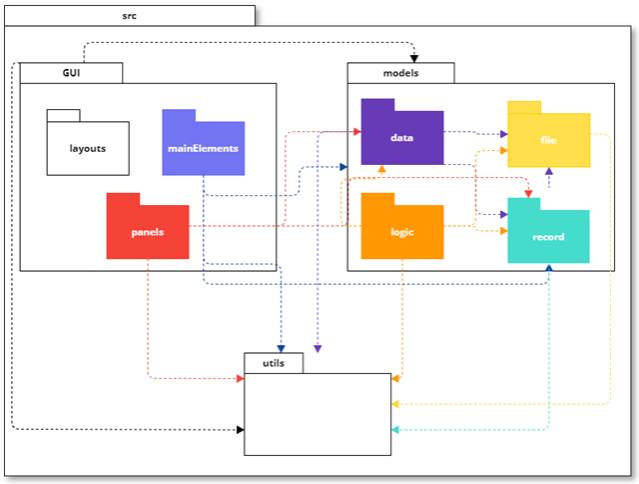

Figure 1: Struttura dei package dell'applicazione.

3 Strutture Dati e scelte Algoritmiche

I dati vengono salvati in modo definitivo in un file `.csv`. Quando si apre l'applicazione, i dati contenuti nei file vengono letti e caricati in delle mappe strutturate come relazioni in un database (chiave primaria + tupla), da cui poi estrarre i dati quando richiessto.
Abbiamo scelto di utilizzare le `HashMap` per diversi motivi. In primo luogo, l'uso di una `HashMap` in memoria offre un accesso rapido ed efficinete ai dati. Una volta che i dati sono stati letti dai file, vengono memorizzati in memoria sottoforma di `HashMap`, consentendo operazioni di accesso rapido con complessità

$O(1)$.
Lavorare direttamente sui file avrebbe richiesto operazioni di lettura e scrittura più frequenti, coinvolgendo l'I/O del disco, che è generalmente più lento rispetto all'I/O in memoria. Memorizzare i dati in una `HashMap` consente di evitare la necessità di aprire e chiudere frequentemente i file durante le operazioni.
Inoltre, le `HashMap` forniscono una struttura dati efficiente per la gestione di associazioi chiave-valore. Questo è particolarmente utile nel nostro caso, dova abbiamo bisogno di eseguire operazioni di ricerca, inserimento e aggiornamento dei dati in modo veloce ed efficiente.
Alcune della altre strutture dati potrebbero essere meno adatte alle esigenze specifiche del progetto. Ad esempio, le liste potrebbero risultare inefficaci per le operazioni di ricerca rapida, mentre altre strutture dati complesse, come alberi, grafi o liste di liste, potrebbero aggiungere una complessità non necessaria alla nostra implementazione.

Ora verranno descritte le sclete algoritmiche e le loro complessità della parte `Model` del progetto, in quanto è la sezione che si occupa della gestione dei dati.

DataStorage

1. Creazione delle Mappe:
    - Inizializzazione nel costruttore tramite i metodi `createCityMap(), createOperatorMap(), createCenterMap(), e createWeatherMap()`.
    - Ogni metodo legge i dati dal file corrispondente utilizzando `FileHandler` e costruisce oggetti (es. `RecordCity, RecordOperator`, ecc.) per l'inserimento nella mappa corrispondente.
    - Complessità: $O(n)$, dove n è il numero di record nel file.
2. Operazioni di Ricerca:
    - Fornisce metodi (es. `getCityByID, getOperatorByID, getCenterByID, getWeatherByID`) per recuperare oggetti specifici dalla mappa basandosi sull'ID.
    - Complessità: $O(1)$, per accesso in una `HashMap`.

DataQuery

1. Metodi di Recupero per Singole Entità:
    - Utilizzando il metodo privato`filterData` per filtrare i dati basandosi su condizioni specifiche.
    - Complessità: $O(n)$, dove n è il numero di elementi nella collezione.
2. Metodi di Filtraggio Interni:
    - Verificano se un elemento soddisfa una condizione specifica.

- Complessità: $O(1)$, ma totale dipende dal numero di condizioni.

3. Generazione di Epsilon:
   - Il metodo `generateEpsilon` calcola un valore di ϵ basato sul numero di posizioni deciamli di un dato valore.
   - Complessità: $O(n)$, dove n è il numero di posizioni decimali.
4. Caloclo delle Posizioni Decimali:
   - Il metodo `computeDecimalPositions` calcola il numero di posizioni decimali di un valore `double`.
   - Complessità: $O(n)$, dove n è la lunghezza della stringa rappresentante il valore.

DataHandler

1. Generazione della Chiave Primaria:
   - Il metodo `generatePrimaryKey` cerca iterativamente la chiave k più alta della mappa attuale e genera una nuova chiave $k' = k + 1$.
   - Complessità: $O(n)$, dove n è il numero di elementi nella mappa.
2. Aggiunta di nuovi Record Operatore, Centro e dati Meteorologici:
   - Il metodo `addNewRecord` genera un nuovo record, verifica se ci sono duplicati, aggiunge il record nella relativa mappa e scrive i dati nel file corrispondente.
   - Complessità: $O(n)$, dove n è il numero di record già presenti.

   Aggiornamento di Record:
   - Il metodo `updateRecord` ricerca il record nel file, lo aggiorna e riscrive il file con il nuovo record.
   - Complessità: $O(n)$, dove n è il numero totale di righe nel file.

LogicCenter

1. Inizializzazione di un nuovo Centro di Monitoraggio:
   - Algorimto con controlli su autenticazione, associazione dell'utene a un centro esistente, validità parametri del nuovo centro e verifica validità ID città.
   - Aggiorna i dati dell'operatore quando viene associato al nuovo centro.
   - Complessità: $O(n)$, dove n è il numero di città associate al centro.
2. Aggiunta di dati meteorologici a un centro:
   - Controlli sull'autenticazione dell'utente, associazione a un centro, validità della data e dei dati meteorologici.

- Aggiorna i dati della mappa e del file corrispondente aggiungendo un nuovo record di dati meteorologici.
- Complessità: $O(n)$, dove n è il numero di righe di dati meteorologici fornite.

LogicCity

1. Costruttore `WeatherTableData`:
   - Construttore con array di record di dati meteorologici e chiamata a `processCategory` per ogni categoria presente nei record.
   - Complessità:$O(n*m)$, dove n è la lunghezza dell'array e m è il numero di categorie.
2. Metodo `processCategory`:
   - Aggiorna i punteggi, i conteggi dei record e i commenti per la categoria data.
   - Complessità: $O(1)$ per ogni chiamata.
3. Metodo `getCategoryAvgScore`:
   - Calcola la media dei punteggi per una categoria data.
   - Complessità: $O(1)$.
4. Metodo `getCategoryRecordCount`:
   - Ottiene il conteggio dei record per una categoria data.
   - Complessità: $O(1)$.
5. Metodo `getCategoryComments`:
   - Ottine la lista di commenti per una categoria data.
   - Complessità: $O(1)$.

LogicOperator

1. Algoritmo di Login `performLogin`:
   - Verifica se i campi per il login non sono vuoti, esegue il `logout` se un operatore è già loggato, costruiisce la lista di condizioni per la query da effettuare e la esegue per ottenere l'operatore corrispondente alle credenziali fornite.
   - Complessità: $O(n)$, dove n è il numero di operatori registrati.
2. Algoritmo di Registrazione `performRegistration`:
   - Controlli di validità su nome, cognome, codice fiscale, e-mail, username e password.

- Aggiunge un nuovo record alla mappa corrispondente.
- Complessità: $O(1)$, in quanto le operazioni sono indipendenti dal numero di operatori registrati.

3. Algoritmo di Associazione a un Centro `associateCenter`:
   - Verifica se l'operatore è loggato ed aggiorna l'operatore corrente con l'ID del centro specificato.
   - Complessità: $O(1)$, in quanto coinvolge solo operazioni di aggiornamento nella mappa.
4. Algoritmo di Validazione dei Dati:
   - Controlli di validità sui dati froniti.
   - Complessità: $O(n)$, dove n è il numero di campi da controllare.
5. Algoritmo di Hashing della Password `hashPassword`:
   - E' utilizzato l'algoritmo `SHA-256` combinato con la concatenazione di username e password.
   - Complessità: $O(n)$, dove n è la lunghezza dell'input in bit.

RecordCity, RecordOperator, RecordCenter, RecordWeather

1. Costruttore del Record:
   - Assegna i valori forniti ai campi del record.
   - Complessità: $O(1)$.
2. Metodo `toString`:
   - Restituisce una rappresentazione testuale del record.
   - Complessità: $O(n)$, dove n è la lunghezza dell'array dei campi.

CurrentOperator

1. Pattern Singleton:
   - Implementato con una variabile statica privata `instance` e costruttore privato.
   - `getInstance()` restituisce l'instanza unica di `CurrentOperator`.
   - Complessità: $O(1)$, coinvolge solo operazioni di controlloe e allocazione di memoria.
2. Pattern Observer:
   - Metodi `addCurrentUserChangeListener` e `removeCurrentUserChangeListener` gestiscono l'aggiunte e la rimozione di listener per i cambiamenti di utente. $O(1)$ per entrambi.

- Metodo `notifyCurrentUserChange` notifica tutti i listener registrati di un cambio di utente. $O(n)$, dove n è il numero di listener registrati.
- Metodo `setCurrentOperator` imposta l'operatore corrente e notidica i listener solo se l'operatore è diverso da quello corrente. $O(n)$, per la chiamata a `notifyCurrentUserChange`.

4 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 2: Costruttore della classe CurrentOperator

Questo pattern è stato implementato seguendo questo schema:

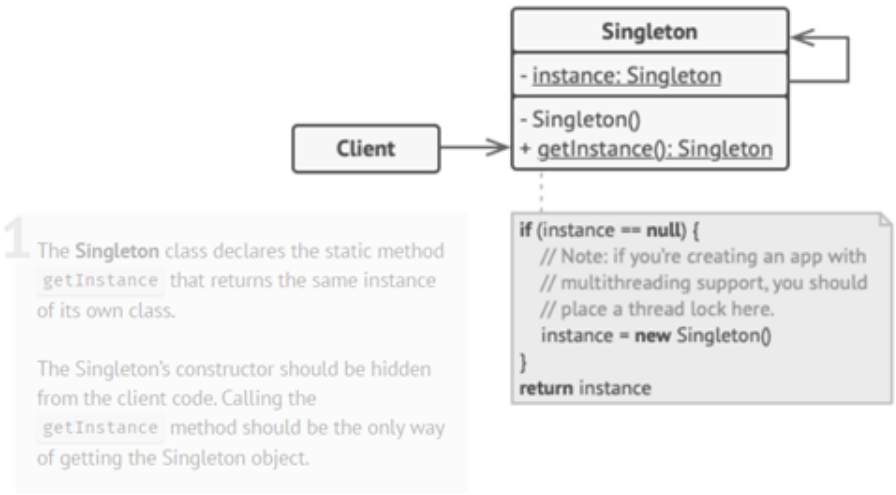

Figure 3: Schema del pattern Singleton

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 4: 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.

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

Figure 5: 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:

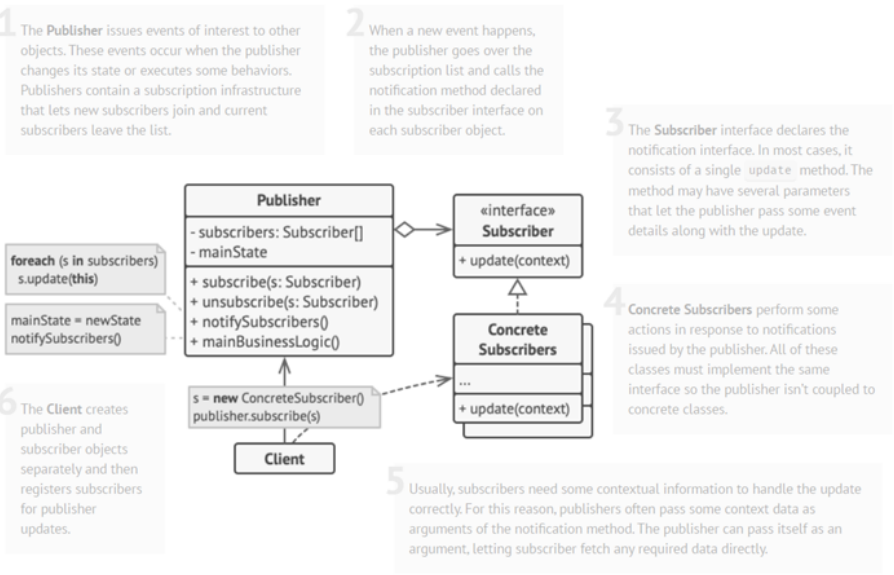

Figure 6: Schema del pattern Observer