Centri Vaccinali
Manuale Tecnico

Version 1.0.0

Guarini Nicolas - 745508
Rizzo Domenico - 745304
Kokaj Redon - 744959
Alzati Filippo - 745495

August 29, 2021

Contents

1 Introduzione

"Centri Vaccinali" è un progetto sviluppato durante il corso di Laboratorio Interdisciplinare A per il corso di Laurea Triennale in Informatica dell'Università degli Studi dell'Insubria, e il linguaggio utilizzato per lo sviluppo dell'applicazione è Java alla sua versione SE 15.
Si tratta di un'applicazione stand-alone, che quindi non necessita di alcun tipo di installazione nel Sistema Operativo, tuttavia utilizza dei file di salvataggio per la gestione dei vari dati utilizzati.

1.1 Librerie esterne utilizzate

Al fine di evitare possibili problemi di compatibilità cross-platform e mettere alla prova le conoscenze e abilità acquisite durante il corso di laurea, si è scelto di non utilizzare alcuna libreria esterna, ma di utilizzare esclusivamente i packages e le relative classi presenti nella libreria standard Java, scrivendo quindi "da zero" tutte le funzionalità non previste dalla stessa.

1.2 Struttura generale del sistema di classi

Il progetto si divide in due packages: `centrivaccinali` e `cittadini`.
Come è facile intuire dal nome, il primo modella e gestisce la parte del programma dedicata ai centri vaccinali, mentre il secondo quella dedicata ai cittadini.
E' importante specificare che il punto di avvio del programma, quindi il metodo `main()`, è contenuto nel package `centrivaccinali` nella classe `CentriVaccinali`.
All'interno di ogni package le classi adibite alla gestione dell'interfaccia grafica sono separate in un'apposita cartella chiamata `UI`, col fine di mantenere la struttura del progetto ordinata e chiara, ma soprattutto per effettuare una vera e propria scissione tra frontend e backend, anche in vista di una futura implementazione di un database.

2 Classi Principali

Le classi principali del progetto sono quelle che si occupano della vera e propria elaborazione dei dati, che quindi prelevano i dati dai file di salvataggio, li elaborano, e li restituiscono all'interfaccia grafica o viceversa, fungendo come un vero e proprio backend dell'applicazione.

2.1 CentriVaccinali

`CentriVaccinali` è la classe che contiene il metodo `main()` e quindi il punto di avvio del programma. E' una classe molto importante perchè contiene tutti quei metodi fondamentali per il funzionamento della sezione dedicata ai Centri Vaccinali, e sui quali si appoggia l'interfaccia grafica per l'elaborazione dei dati.

2.1.1 Complessità stimate

Di seguito una lista dei metodi della classe `CentriVaccinali` con una breve analisi di complessità:

- `registraCentroVaccinale()`:
  - Chiama il metodo `getCentriVaccinali()` e salviamo la lista che esso ritorna
    - Complessità in tempo: $O(n)$
    - Complessità in spazio: $O(n)$
  - Aggiunge il centro vaccinale da registrare alla lista
    - Complessità in tempo: $O(1)$ (peculiarità delle LinkedList)
    - Complessità in spazio: $O(1)$
  - Serializza la lista aggiornata chiamando `serializzaCentriVaccinali()`
    - Complessità in tempo: $O(n)$
    - Complessità in spazio: $O(1)$

  Complessità totale del metodo in tempo: $O(n)$
  Complessità totale del metodo in spazio: $O(n)$

- `getCentriVaccinali()`
  Ritorna la lista frutto della deserializzazione del contenuto del file
  - Complessità in tempo: $O(n)$
  - Complessità in spazio: $O(n)$ (viene istanziata una lista di dimensione n)

- `serializzaCentriVaccinali()`
  - Viene istanziato un oggetto di tipo `ObjectOutputStream`
    - Complessità in tempo: $O(1)$
    - Complessità in spazio: $O(1)$

  - Viene serializzata la lista su file
    * Complessità in tempo: $O(n)$
    * Complessità in spazio: $O(1)$

  Complessità totale del metodo in tempo: $O(n)$
  Complessità totale del metodo in spazio: $O(1)$

- `registraVaccinazione()`: analogo a `registraCentroVaccinale()`
- `getVaccinazioni()`: analogo a `getCentriVaccinali()`
- `serializzaVaccinazioni()`: analogo a `serializzaCentriVaccinali()`
- `getNumSegnalazioniEventiAvversi()`
  - Viene chiamato il metodo `getVaccinazioni()` e viene salvata la lista che ritorna
    * Complessità in tempo: $O(n)$
    * Complessità in spazio: $O(n)$
  - Per ogni elemento della lista si salva la dimensione della lista `eventiAvversi`
    * Complessità in tempo: $O(n)$
    * Complessità in spazio: $O(1)$
  - La somma di tutte le dimensioni (salvata nella variabile `count` viene ritornata

  Complessità totale del metodo in tempo: $O(n)$
  Complessità totale del metodo in spazio: $O(n)$

- `getSeveritaMediaEventiAvversi()`:
- Viene chiamato il metodo `getVaccinazioni()` e viene salvata la lista che ritorna
  - Complessità in tempo: O(n)
  - Complessità in spazio: O(n)
- Vengono effettuati due cicli annidati: per ogni vaccinazione si cicla la lista degli eventi avversi, eseguendo operazioni di costo $O(1)$ nel ciclo più interno.
  Con n = numero di vaccinazioni e m = somma della dimensione della lista `eventiAvversi` di ogni vaccinazione, le complessità sono le seguenti:

  - Complessità in tempo: $O(n^m)$
  - Complessità in spazio: (n)

Complessità totale del metodo in tempo: $O(n^m)$
Complessità totale del metodo in spazio: $O(n)$

2.2 Cittadini

La classe `Cittadini` contiene i metodi fondamentali per il funzionamento della sezione dedicata ai Cittadini, e sui quali si appoggia l'interfaccia grafica per l'elaborazione dei dati.

2.2.1 Complessità stimate

Di seguito una lista dei metodi della classe `Cittadini` con una breve analisi di complessità:

- `registraCittadino()`: analogo a `registraCentroVaccinale()`
- `getCittadini()`: analogo a `getCentriVaccinali()`
- `serializzaCittadini()`: analogo a `serializzaCentriVaccinali()`
- `sha256()`
  - Complessità in tempo: $O(n)$ dove n è la lunghezza della password da crittografare
  - Complessità in spazio: $O(1)$
- `registraEventoAvverso()`:
  - Chiamato il metodo `getVaccinazioni()` e salvata la lista da esso ritornata
    * Complessità in tempo: $O(n)$
    * Complessità in spazio: $O(n)$
  - Si scandisce la lista fino a quando non si trova il cittadino desiderato. Una volta trovato verrà aggiunto alla lista `eventiAvversi` l'evento avverso da aggiungere
    * Complessità in tempo: $O(n)$
    * Complessità in spazio: $O(1)$
  - Viene serializzata la lista aggiornata di vaccinazioni

    * Complessità in tempo: $O(n)$
    * Complessità in spazio: $O(1)$

  Complessità totale del metodo in tempo: $O(n)$
  Complessità totale del metodo in spazio: $O(n)$

3 Classi che modellano la realtà di interesse

Le classi che modellano la realtà di interesse sono quelle classi che non effettuano particolari operazioni di calcolo o sui dati, bensì rappresentano e modellano degli oggetti facenti parte della realtà di interesse, come per esempio un indirizzo o un cittadino, replicando le loro caratteristiche e peculiarità rilevanti.

3.1 CentroVaccinale

Modella le caratteristiche di un centro vaccinale, con le seguenti proprietà:

- `String nome`
- `String tipologia`
- `Indirizzo indirizzo`
- `String id`: codice di 5 cifre che identifica il centro vaccinale

3.2 EventoAvverso

Modella le caratteristiche di un evento avverso, ovvero un effetto collaterale, con le seguenti proprietà:

- `String nome`
- `String noteAggiuntive`
- `int severita`

3.3 Indirizzo

Modella le caratteristiche di un indirizzo, con le seguenti proprietà:

- `String qualificatore`

- `String nome`
- `String numeroCivico`
- `String comune`
- `String provincia`
- `String CAP`

Il numero civico è rappresentato da una stringa, invece di un intero, per poter gestire anche gli indirizzi che contengono una lettera nel numero civico (ES: via Cavour 8/E).
Il CAP, invece, è rappresentato da una stringa invece di un intero, come sarebbe facile pensare dato che è composto da soli numeri, in quanto esistono svariati CAP in cui i primi numeri sono degli zeri (per esempio il CAP di Roma è 00184), e in quei casi verrebbero troncati e quindi al posto di salvare "00184" si salverebbe "184". Utilizzando una stringa, invece, non si hanno questi problemi.

3.4 Vaccinazione

Modella le caratteristiche di una somministrazione, con le seguenti proprietà:

- `String nome, cognome, cf, id, nomeVaccino`
- `Date data`
- `CentroVaccinale centroVaccinale`
- `LinkedList<EventoAvverso> eventiAvversi`

3.5 Cittadino

Modella le caratteristiche di un cittadino, con le seguenti proprietà:

- `String nome, cognome, cf, email, username, password, idVaccinazione`
- `CentroVaccinale centrovaccinale`

E' bene specificare che la password con cui l'utente si è registrato non viene mai salvata in chiaro, ma viene salvato il suo hash. E' possibile trovare maggiori informazioni nell' approfondimento dedicato.

4 Classi per la gestione dell'interfaccia grafica

Le classi adibite alla gestione dell'interfaccia grafica si occupano di creare e gestire il ciclo di vita delle schermate con cui l'utente interagisce.
Nello specifico, si è scelto di utilizzare il framework Swing, in quanto si adatta molto bene ai diversi Sistemi Operativi (cross-platform), è semplice, veloce, ed è presente nella libreria standard di Java.
Tuttavia, non essendo l'interfaccia grafica espressamente richiesta nei requisiti di progetto, non verrà effettuata una discussione dettagliata delle singole classi, ma verranno solamente citate nell'elenco di seguito.

4.1 Sezione Centri Vaccinali

- `UIStartMenu`
- `UICentriVaccinali`
- `UIVisualizzaCentriVaccinali`
- `UIRegistraCentroVaccinale`
- `UIRegistraVaccinato`

4.2 Sezione Cittadini

- `UICittadini`
- `UICercaCentriVaccinali`
- `UIInfoUtente`
- `UILoginCittadino`
- `UIRegistraCittadino`
- `UISegnalaEventoAvverso`
- `UIUtenteLoggato`

5 Strutture dati utilizzate

Durante lo sviluppo dell'applicazione non è risultato necessario implementare particolari strutture dati.
Le uniche strutture dati utilizzate (oltre ovviamente a quelle elementari come stringhe e interi) sono le liste. Nello specifico si è optato per delle LinkedList.

5.1 LinkedList

Dato che si sarebbe dovuto avere a che fare con delle serie di dati (una serie di vaccinazioni, una serie di centri vaccinali, una serie di cittadini registrati ...) era ovvio che si sarebbero dovute usare delle strutture dati adatte come degli array o delle liste.
Dato che la dimensione di queeste collezioni di oggetti sarebbe dovuta cambiare dinamicamente durante l'esecuzione del programma si è scelto di implementare delle liste.
Il quesito a questo punto era quale tipo di liste utilizzare, e le alternative erano le LinkedList o le ArrayList.
Per entrambe le implementazioni, le relative operazioni hanno costo $O(n)$, con alcune differenze: nelle LinkedList, essendo delle double-linked list, l'accesso al primo e all'ultimo elemento ha costo $O(1)$, e quindi anche inserimento e rimozione (alla prima o all'ultima posizione) hanno costo $O(1)$. Nelle ArrayList, invece, queste operazioni hanno costo $O(n)$.
Dato che l'operazione di `add()` è molto comune all'interno del progetto si è deciso di utilizzare le `LinkedList`.
Tuttavia, `LinkedList` e `ArrayList` hanno prestazioni molto simili, dato che `ArrayList` compensa lo svantaggio del costo di inserimento con il costo dell'operazione `get()`, che ha costo $O(1)$, contro il costo di $O(n)$ delle `LinkedList` per la stessa operazione.
In linea di massima, quindi, entrambe le implementazioni sarebbero risultate valide.

5.2 ID Vaccinazione

L'ID vaccinazione è composto da 16 cifre: le prime 5 identificano il centro vaccinale, e le restanti 11 identificano il vaccinato. In questo modo dall'ID è possibile risalire al centro vaccinale dove è stata effettuata la vaccinazione e quindi risalire al vaccinato cercandolo dentro al file relativo alle vaccinazioni per quel centro vaccinale.
In questo modo non vi è la necessità di chiedere all'utente in quale centro vaccinale si è recato per effettuare la vaccinazione.

6 Gestione e Formato dei File

Come da specifiche di progetto, il file di salvataggio dei dati sono contenuti nella cartella `data/` e nominati con il formato "NomeFile.dati.txt".
Ci sono tre tipi di file di salvataggio: quelli dei centri vaccinali, quelli dei cittadini registrati, e quelli relativi alle vaccinazioni di uno specifico centro vaccinale.
Per ragioni di sicurezza e di velocità della soluzione le liste che contengono gli oggetti da salvare non sono salvate in chiaro (come succede con il formato `.csv`) ma sono serializzate.

6.1 Salvataggio delle password

Particolare attenzione è stata dedicata al salvataggio delle password. Infatti, anche serializzando gli oggetti prima di salvarli su file, una persona esperta riuscirebbe a risalire alle password senza troppe difficoltà. Per questo si è scelto di non salvare mai nessuna password in chiaro, bensì di effettuare l'hashing con l'algoritmo `SHA-256`, che è tra i più diffusi e sicuri, e salvare il risultato sotto forma di stringa nel formato `base64`.
Nel momento del login, la password inserita nel form subirà lo stesso processo di hashing e verrà poi confrontata con la password salvata relativa a quell'utente, e se i due hash corrispondono le due password iniziali coincidono e il login è autorizzato.

6.2 Scrittura su file

Il processo di scrittura su file è gestito da un metodo che prende come parametro la lista di oggetti da serializzare, la serielizza e la scrive su file, utilizzando le classi `FileOutputStream` e `ObjectOutputStream`.
Di seguito un esempio con il metodo `serializzaCittadini()`:

```
static void serializzaCittadini(LinkedList<Cittadino>
↪  cittadini){
    try{
        FileOutputStream fileOutputStream = new
        ↪  FileOutputStream("./data/Cittadini_Registrati.dati.txt");
        ObjectOutputStream objectOutputStream = new
        ↪  ObjectOutputStream(fileOutputStream);
        objectOutputStream.writeObject(cittadini);
        objectOutputStream.close();
    }catch(IOException e){
        e.printStackTrace();
```

```
    }
}
```

6.3 Lettura da file

Il processo di lettura da file è analogamente inverso: legge da file, converte il contenuto nell'oggetto adatto, e lo ritorna, utilizzando le classi `FileInputStream` e `ObjectInputStream`. Di seguito un esempio con il metodo `getCittadini()`:

```
public static LinkedList<Cittadino> getCittadini(){
    LinkedList<Cittadino> cittadini = new LinkedList<>();

    try{
        File fileCittadini = new
        ↪  File("./data/Cittadini_Registrati.dati.txt");
        if(fileCittadini.createNewFile()){
            serializzaCittadini(new LinkedList<>());
        }else{
            FileInputStream fileInputStream = new
            ↪  FileInputStream(fileCittadini);
            ObjectInputStream objectInputStream = new
            ↪  ObjectInputStream(fileInputStream);
            cittadini = (LinkedList<Cittadino>)
            ↪  objectInputStream.readObject();
        }
    }catch(IOException | ClassNotFoundException e){
        e.printStackTrace();
    }

    return cittadini;
}
```

Nel caso in cui il file non esista, oltre ad essere creato, viene anche inizializzato con una lista vuota. In questo modo si evitano situazioni patologiche future.

7 Bibliografia e citazioni

- Stackoverflow, Online: https://stackoverflow.com/questions/322715/when-to-use-linkedlist-over-arraylist-in-java
- Crypto Stackexchange, Online: https://crypto.stackexchange.com/questions/67448/what-is-the-time-complexity-of-computing-a-cryptographic-hash-function-random-or