Centri Vaccinali
Manuale Tecnico

Version 2.0.0

Guarini Nicolas - 745508
Rizzo Domenico - 745304

August 25, 2022

Contents

1 Introduzione

"Centri Vaccinali" è un progetto sviluppato durante il corso di Laboratorio Interdisciplinare B 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 8 (o 1.8), mentre come sistema di gestione progetto e build è stato utilizzato Maven alla sua versionec 3.8.6. Si tratta di un'applicazione stand-alone, che quindi non necessita di alcun tipo di installazione nel Sistema Operativo e porta con se tutte le sue dipendenze. Il progetto è stato sviluppato e testato su Windows 11, Windows 10 e Linux (Manjaro), pertanto non si garantisce il funzionamento su MacOS.

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 prediligere librerie native e quello non presente svilupparlo da zero. Tuttavia alcune dipendenze esterne si sono rivelate inevitabili:

- `maven-jar-plugin` (sia per l'applicazione client sia per quella server): necessaria per garantire il corretto funzionamento delle funzionalità di build e gestione dei manifest di Maven.
- `postgresql` (solo applicazione server): dipendenza necessaria per potersi interfacciare correttamente al database.
- `maven-javadoc-plugin`: dipendenza necessaria per la generazione a tempo di build della documetazione javadoc
- `maven-shade-plugin`: dipendenza necessaria per effettuare l'operazione di *package* comprendendo nel `.jar` tutte le dipendenze.

1.2 Struttura generale del sistema di classi

Il progetto si articola in tre moduli:

- `clientCV`: contiene il codice relativo all'applicazione client
- `serverCV`: contiene il codice relativo all'applicazione server
- `common`: contiene tutti gli artifatti che vengono utilizzati da entrami i moduli sopra, in modo da evitare ridondanze di dati e conseguenti problemi di versionamenti degli stessi.

Il punto di avvio dell'applicazione client è contenuto nella classe `clientCV.centrivaccinali.CentriVaccinali`, mentre quello dell'applicazione server è contenuto nella classe `serverCV.ServerCV`
All'interno di ogni package le classi adibite alla gestione dell'interfaccia grafica sono separate in un apposito package chiamato `UI`, col fine di mantenere la struttura del progetto ordinata e chiara, ma soprattutto per effettuare una scissione tra logica di presentazione e logica applicativa.

Di seguito sono riportati alcuni diagrammi UML per fornire una visione d'insieme più completa e chiara sulle classi e le funzionalità dell'applicazione.

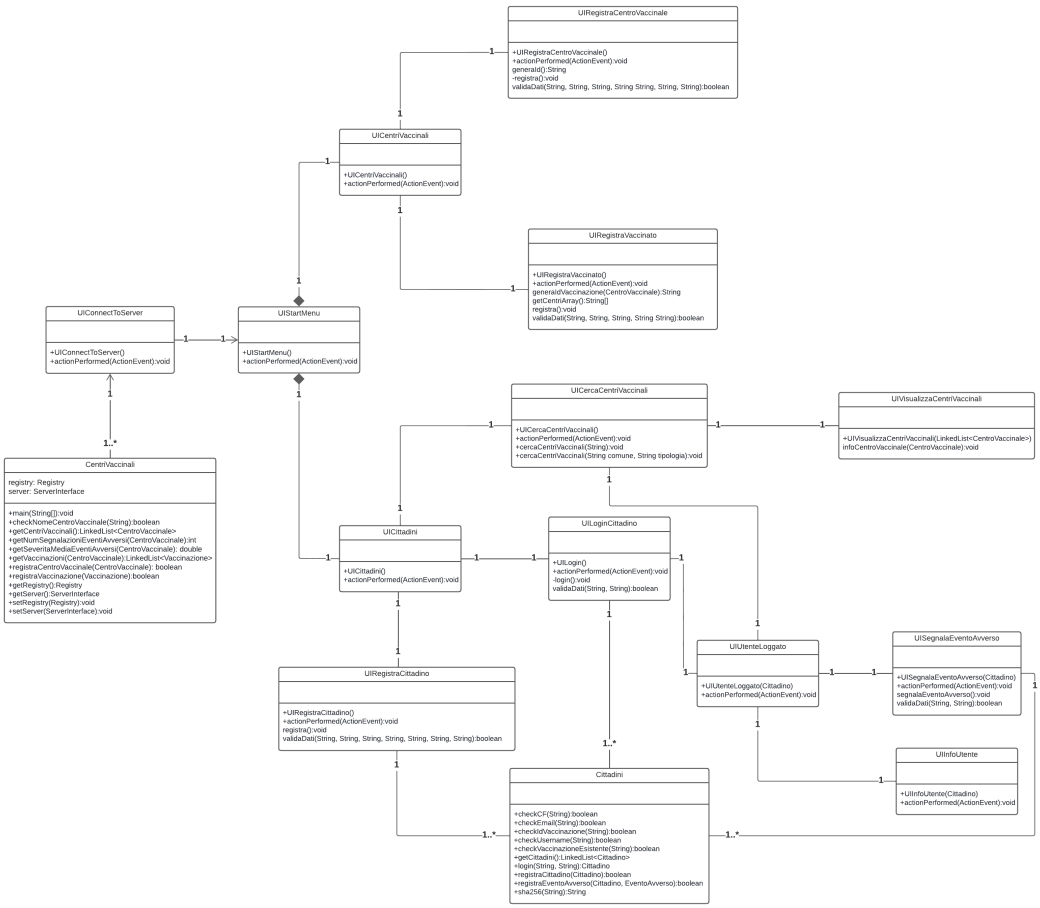

Figure 1: Class Diagram - Client

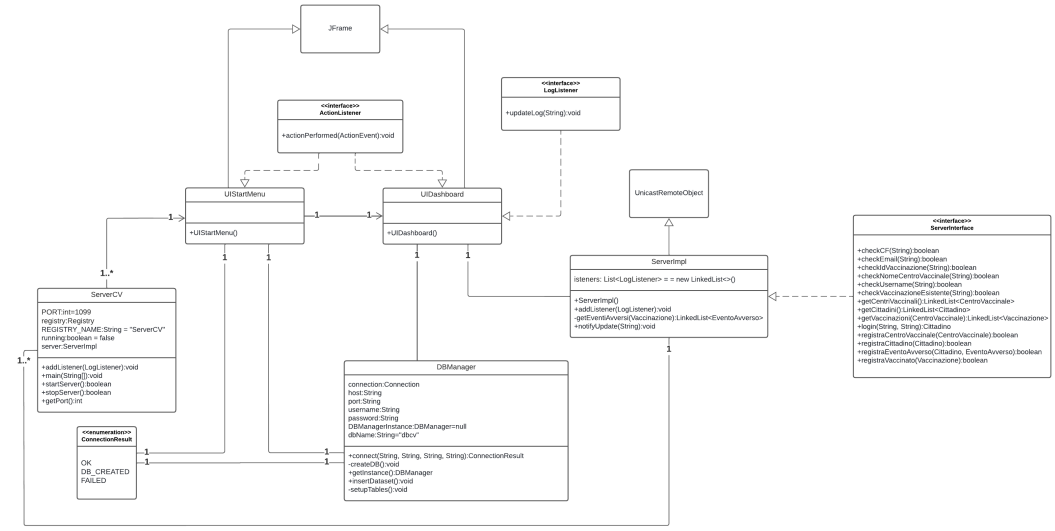

Figure 2: Class Diagram - Server

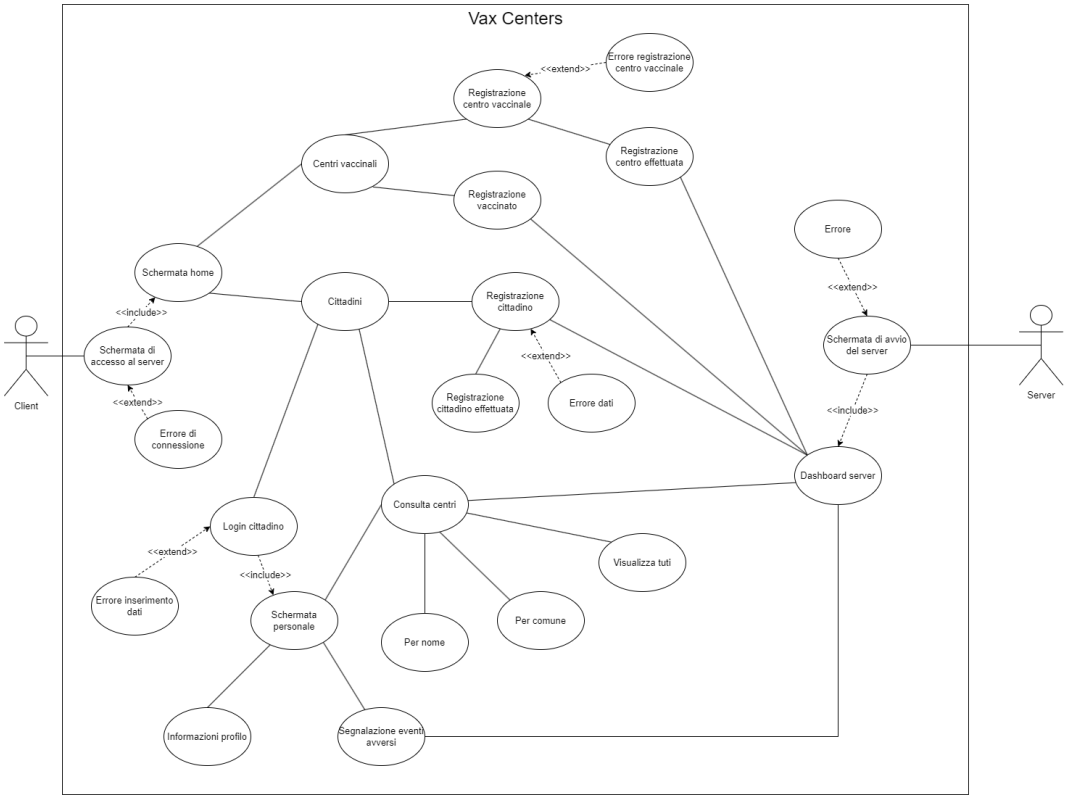

Figure 3: Use Case Diagram

1.3 Compilare da sorgente

1.3.1 Installare Maven

Dato che il progetto si basa su Maven come sistema di build e gestione dipendenze, è necessario avere un'installazione attiva di Maven (versione consigliata: $\geq$3.8.6) che possibile scaricare facilmente dal sito ufficiale.

E' inoltre necessario aggiungere il percorso della cartella contenente i binari di Maven all'interno del PATH, in modo da poter eseguire i comandi da qualsiasi directory del sistema operativo.

Per verificare la corretta installazione di Maven si esegui il comando `mvn --version` nel terminale. Si dovrebbe ottenere un output simile al seguente.

```
Apache Maven 3.8.6 (84538c9988a25aec085021c365c560670ad80f63)
Maven home: C:\apache-maven-3.8.6
Java version: 15, vendor: Oracle Corporation, runtime:
↪  C:\Program Files\Java\jdk-15
Default locale: it_IT, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family:
↪  "windows"
```

1.3.2 Compilare con IntelliJ

IntelliJ, così come la maggior parte degli IDE specializzati in progetti Java, riconosce automaticamente un progetto Maven, e mette a disposizione un menù dedicato per gestire tutti i processi di build (lifecycle) da interfaccia grafica.

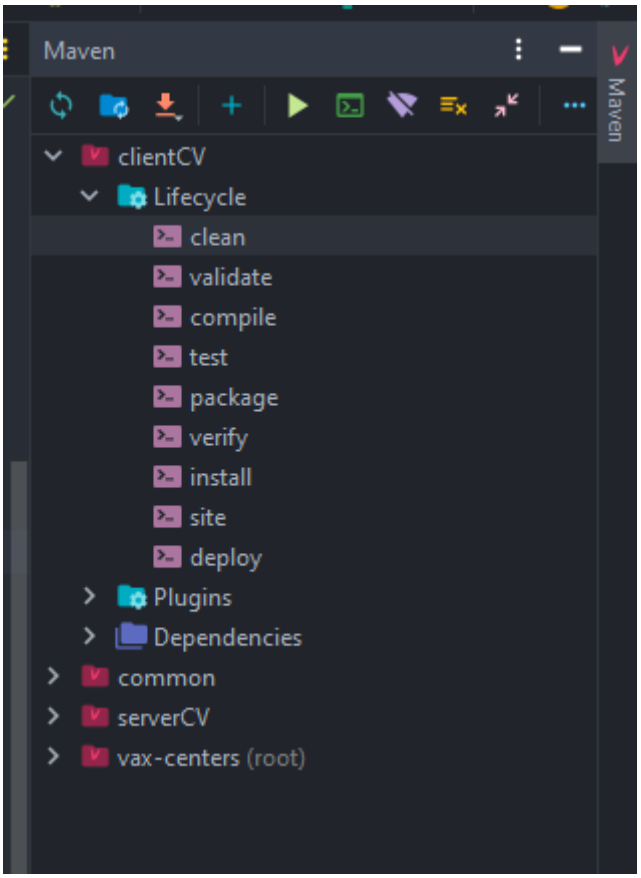

Figure 4: Maven Lifecycle

I metodi lifecycle consigliati per compilare l'applicazione sono i seguenti, da eseguire nell'ordine in cui sono riportati:

- clean
- validate
- compile
- package

Una volta eseguiti, nei moduli sarà stata generata una cartella `target/` che conterrà, oltre ai file .class compilati, anche l'eseguibile di quel modulo, nominato in questo modo: `clientCV-2.0.jar` e `serverCV-2.0.jar`

Per quanto riguarda la generazione della documentazione javadoc, è necessario eseguire il plugin `javadoc:javadoc`, e verrà generata all'interno della cartella `target/site/apidocs/` di ogni modulo.

1.3.3 Compilare da CLI

Per compilare da terminale sarà necessario eseguire i seguenti comandi (NB: è necessario che ci si trovi nella root del progetto):

- `mvn clean`
- `mvn validate`
- `mvn compile`
- `mvn package`

Gli eseguibili si troveranno all'interno della cartella `target/` di ogni modulo.

Per generare la documentazione javadoc, il comando da eseguire è il seguente:

- `mvn javadoc:javadoc`

La documentazione sarà generata all'interno della cartella `target/site/apidocs/` di ogni modulo.

2 Database

2.1 Realtà di interesse

Per questo progetto era necessario implementare un database relazionale in grado di gestire le informazioni relative a centri vaccinali, vaccinazioni, cittadini e segnalazione di eventi avversi. Nello specifico, gli operatori dei centri vaccinali dovranno poter registrare nuovi centri vaccinali e registrare le vaccinazioni effettuate all'interno di essi, mentre i cittadini potranno controllare statistiche e informazioni sui centri vaccinali e, una volta registrati usando l'id fornito dopo la vaccinazione, potranno segnalare eventuali eventi avversi notati a seguito della somministrazione. Il diagramma ER prodotto in seguito all'analisi dei requisiti è il seguente.

2.2 Diagramma ER

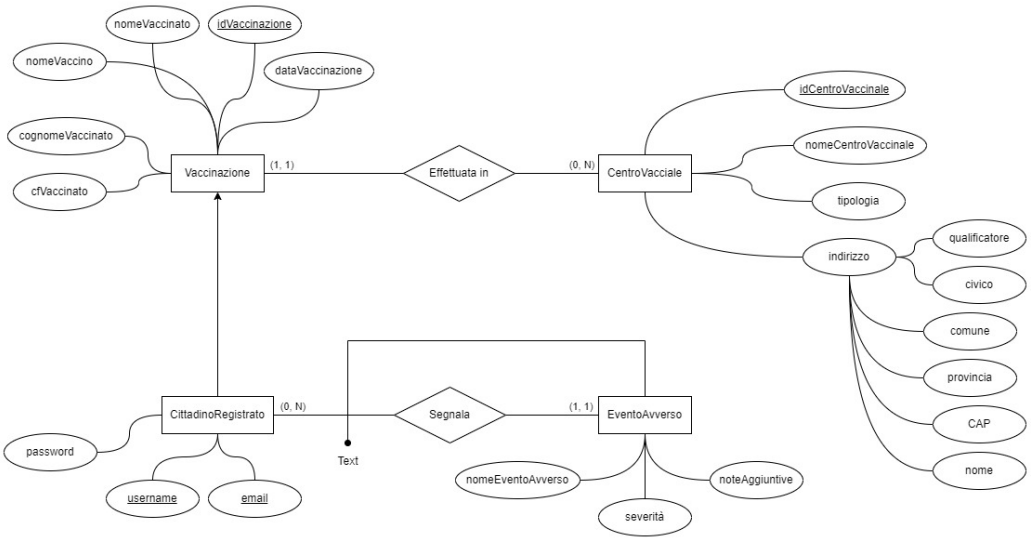

Figure 5: Diagramma ER

2.3 Ristrutturazione

- L'attributo composto `Indirizzo` è stato scomposto eliminando i sotto-attributi. Sarà quindi compito dell'applicazione garantire che il nuovo attributo contenga valori coerenti con la semantica dell'attributo composto ristrutturato. Il motivo per il quale si è scelta questa opzione piuttosto che scegliere di considerare tutti i sotto-attributi come attributi dell'entità `CentroVaccinale` è che l'applicazione utilizza le informazioni riguardanti gli indirizzi sempre in modo sommario, e quindi sarebbe uno spreco di memoria creare 6 nuovi attributi.
- Per eliminare la generalizzazione tra `Vaccinazione` e `CittadinoRegistrato` si è scelto di optare per l'eliminazione dell'entità figlia e quindi accorparla nell'entità padre, dove verrà aggiunto un nuovo attributo booleano `registrato` che permette di distinguere a quale tipo di entità appartiene ogni occorrenza.

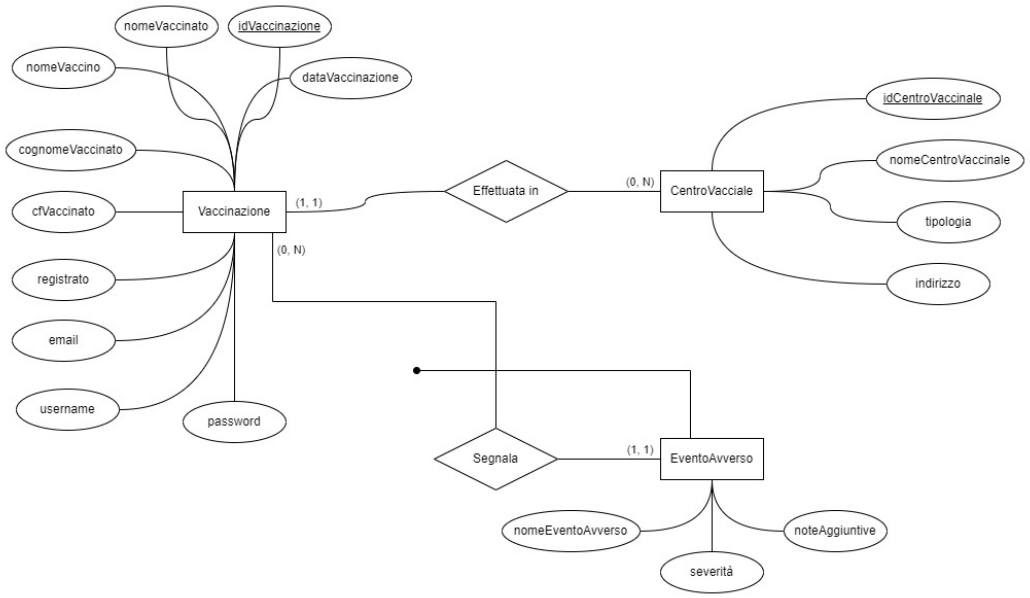

Figure 6: Diagramma ER Ristrutturato

2.4 Traduzione

- Per tradurre l'associazione binaria uno a molti "Effettuata in" tra `Vaccinazione` e `CentroVaccinale`, dato che è presente una partecipazione obbligatoria dal lato uno, si è scelto di tradurre l'associazione mediante attributi, aggiungendo quindi un nuovo attributo `idcentrovaccinale` (chiave esterna su `CentroVaccinale` all'interno dell'entità `Vaccinazione`.
- Per tradurre l'associazione binaria uno a molti "Segnala" tra `Vaccinazione` e `EventoAvverso`, dato che è presente una partecipazione obbligatoria dal lato uno, si è scelto di tradurre l'associazione mediante attributi, aggiungendo quindi un nuovo attributo `idvaccinazione` all'interno dell'entità `EventoAvverso`.
- Per tradurre l'identificatore esterno dell'entità `EventoAvverso` si è scelto di porre come chiave primaria gli attributi `idvaccinazione`, `nomeeventoavverso`

2.5 Vincoli d'integrità

I vincoli d'integrità derivanti dall'analisi dei requisiti e dai processi di ristrutturazione e traduzione sono i seguenti:

- V1: se `registrato=true` allora gli attributi `username, email, password` dell'entità `Vaccinazione` dovranno essere `NOT NULL`

2.6 Creazione tabelle

Lo script SQL utilizzato per la creazione delle tabelle è il seguente:

```sql
CREATE TABLE IF NOT EXISTS CentriVaccinali(
    idCentroVaccinale VARCHAR(5) PRIMARY KEY NOT NULL,
    nomeCentroVaccinale VARCHAR(50) NOT NULL,
    tipologia VARCHAR(20) NOT NULL,
    indirizzo VARCHAR(100) NOT NULL
);

CREATE TABLE IF NOT EXISTS Vaccinazioni(
    idVaccinazione VARCHAR(16) PRIMARY KEY NOT NULL,
    nomeVaccinato VARCHAR(50) NOT NULL,
    cognomeVaccinato VARCHAR(50)NOT NULL,
    cfVaccinato VARCHAR(16) UNIQUE NOT NULL,
    dataVaccinazione DATE NOT NULL,
    idCentroVaccinale VARCHAR(5) NOT NULL,
    nomeVaccino VARCHAR(50) NOT NULL,
    registrato BOOLEAN NOT NULL,
    username VARCHAR(100) NULL,
    email VARCHAR(100) NULL,
    password VARCHAR(64) NULL,
    CONSTRAINT fk_centrivaccinali
        FOREIGN KEY (idCentroVaccinale)
        REFERENCES CentriVaccinali(idCentroVaccinale)
        ON UPDATE CASCADE ON DELETE RESTRICT
);

CREATE TABLE IF NOT EXISTS EventiAvversi(
    nomeEventoAvverso VARCHAR(100) NOT NULL,
    severita SMALLINT NOT NULL,
    noteAggiuntive VARCHAR(5000) NULL,
    idVaccinazione VARCHAR(16) NOT NULL,
    PRIMARY KEY (nomeEventoAvverso, idVaccinazione),
    CONSTRAINT fk_vaccinazione
        FOREIGN KEY (idVaccinazione)
        REFERENCES Vaccinazioni(idVaccinazione)
        ON DELETE CASCADE ON UPDATE CASCADE
);
```

2.7 Queries

Nell'applicazione vengono eseguite numerose query, eccone alcune.

- `getCittadini()`

```
     SELECT * FROM
     (vaccinazioni JOIN centrivaccinali ON
↪    vaccinazioni.idcentrovaccinale =
↪    centrivaccinali.idcentrovaccinale)
     WHERE registrato = true
```

- `login(String username, String password)`

```
    SELECT * FROM
    vaccinazioni JOIN centrivaccinali ON
↪   vaccinazioni.idcentrovaccinale =
↪   centrivaccinali.idcentrovaccinale
    WHERE registrato = true AND username = ? AND password =
↪   ?
```

- `getVaccinazioni(CentroVaccinale centrovaccinale)`

```
    SELECT * FROM vaccinazioni
    WHERE idcentrovaccinale = ?
```

- `registraCittadino(Cittadino cittadino)`

```
    UPDATE vaccinazioni
    SET registrato=true, username=?, password=?, email=?
    WHERE idvaccinazione=?
```

3 Classi Principali

Le classi principali del progetto sono quelle che si occupano della vera e propria elaborazione dei dati, che quindi contattano il server facendosi ritornare i dati, li elaborano, e li restituiscono all'interfaccia grafica, oppure fanno altri tipi di calcoli come per esempio l'hashing delle password, oppure effettuano operazioni importanti come l'avvio e lo stop della connessione RMI.

3.1 CentriVaccinali.java

`clientCV.centrivaccinali.CentriVaccinali` è la classe che contiene il metodo `main()` e quindi il punto di avvio dell'applicazione client. 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. E' composta dai seguenti metodi:

- `registraCentroVaccinale()`
- `getCentriVaccinali()`
- `registraVaccinazione()`
- `getVaccinazioni()`
- `getNumSegnalazioniEventiAvversi()`
- `getSeveritaMediaEventiAvversi()`
- `checkNomeCentroVaccinale()`

Tutti i metodi che si fanno ritornare qualcosa (quindi quelli che iniziano con `get`) hanno complessità in tempo lineare $O(n)$, questo perchè o attendono che il server elabori la query dal database e ricostruisca la lista di n oggetti, o perchè fanno operazioni particolari sulla lista di n oggetti ritornata dal server. L'unica eccezione è il metodo `getSeveritaMediaEventiAvversi()` che, dato che la lista degli eventi avversi è contenuta negli oggetti delle vaccinazioni, per effettuare il calcolo deve eseguire due cicli `for` annidati, che portano la complessità in tempo del metodo a $O(n^m)$, con n=numero di elementi della lista di vaccinazioni, e m=somma della dimensione della lista di eventi avversi di ogni vaccinazione.

I metodi che richiedono di registrare qualcosa nel database o richedono di effettuare dei controlli, invece, hanno complessità $O(1)$ (si può affermare ciò astraendo dall'implementazione del database e ragionando prettamente in termini di codice Java).

3.2 Cittadini.java

La classe `clientCV.cittadini.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. E' composta dai seguenti metodi:

- `registraCittadino()`
- `getCittadini()`

- `sha256()`
- `registraEventoAvverso()`
- `login()`
- `checkUsername()`
- `checkEmail()`
- `checkIdVaccinazione()`
- `checkCF()`
- `checkVaccinazioneEsistente()`

Lo stesso discorso fatto sulle complessità dei metodi della classe `CentriVaccinali` si estende anche ai metodi di questa classe [2].

3.3 ServerCV.java

La classe `serverCV.ServerCV` contiene il punto di avvio dell'applicazione server, e mantiene gli oggetti relativi alla connssione tramite RMI effettuando operazioni vitali per il corretto funzionamento della comunicazione tra client e server. E' composta dai seguenti metodi:

- `startServer()`: imposta la connessione RMI istanziando un nuovo registry e un nuovo oggetto remoto
- `stopServer()`: termina l'attività dell'oggetto remoto e quindi effettua il suo unexport e l'unbind del servizio.
- `addListener()`: aggiunge un nuovo oggetto `LogListener` alla lista dei listener interessati all'aggiornamento dei log (implementazione del pattern Observer che verrà discusso nei capitoli successivi)

3.4 DBManager.java

La classe `serverCV.DBManager` implementando il pattern Singleton, inizializza il database e permette alle altre classi di accedervi (globalmente). Si occupa quindi di connettersi al server postgres, creare (eventualmente) il database, effettuare il setup delle tabelle, e inserire all'interno del database il dataset di test. I suoi metodi sono i seguenti:

- `getInstance()`

- `connect()`
- `createDB()`
- `setupTables()`
- `insertDataset()`

4 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.

4.1 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

4.2 EventoAvverso

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

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

4.3 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.

4.4 Vaccinazione

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

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

4.4.1 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.

4.5 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 (sha256).

5 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. Nella documentazione utente è possibile trovare degli screenshot e spiegazioni dettagliate su ogni schermata.

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.

Ogni modulo del progetto ha il suo package `UI` che contiene le classi che gestiscono l'interfaccia grafia, ecco un elenco di queste ultime:

5.1 Package `clientCV.centrivaccinali.UI`

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

5.2 Package `clientCV.centrivaccinali.UI`

- `UICittadini`
- `UICercaCentriVaccinali`
- `UIInfoUtente`

- `UILoginCittadino`
- `UIRegistraCittadino`
- `UISegnalaEventoAvverso`
- `UIUtenteLoggato`
- `UIVisualizzaCentriVaccinali`

5.3 Package `serverCV.UI`

- `UIStartMenu`
- `UIDashboard`

6 Strutture dati e pattern utilizzati

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 e le enumerazioni. Nello specifico si è optato per delle LinkedList.

Per quanto riguarda i design pattern implementati, sono stati implementati il pattern Singleton per gestire l'istanza del database, e il pattern Observer per gestire l'aggiornamento del pannello contenente i messaggi di log nella schermata `serverCV.UI.UIDashboard`

6.1 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 $O(1)$ [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.

6.2 Enumerazioni

L'unica enumerazione che è risultato necessario implementare è `serverCV.ConnectionResult`:

```
public enum ConnectionResult {
    OK, DB_CREATED, FAILED
}
```

L'utilizzo che ne viene fatto è quello di fornire dettagli sull'esito di connessione al database, dal momento che a ogni login ci si può trovare davanti a tre scenari:

- La connessione al server è andata a buon fine e il database è già esistente
- La connessione al server è andata a buon fine ma il database non esiste e quindi è stato creato
- La connessione al server non è andata a buon fine

Grazie a questa enumerazione, che viene ritornata dal metodo `serverCV.DBManager.connect()`, è possibile comunicare all'utente quello che è successo nel dettaglio.

6.3 Pattern Singleton

Il pattern Singleton è stato implementato nella classe `serverCV.DBManager`. Il suo compito consiste nell'impedire che da una classe possa essere creato più di un oggetto. Per farlo, l'oggetto desiderato viene creato all'interno della classe per poi essere invocato come istanza statica accessibile globalmente [4].

```
public class DBManager{
    //...
    private static DBManager DBManagerInstance = null;
    //...
    public static DBManager getInstance(){
        if(DBManagerInstance == null) DBManagerInstance = new
        ↪  DBManager();
        return DBManagerInstance;
    }
}
```

6.4 Pattern Observer

La necessità di implementare questo pattern nasce dal bisogno di notificare la classe `serverCV.UIDashboard` di un nuovo accesso all'oggetto remoto da parte di un client, in modo da mostrare all'interno del pannello contenente i messaggi di log i dettagli di tale richiesta.

In poche parole, abbiamo un oggetto che viene osservato (l'oggetto remoto `ServerImpl`) e uno o più oggetti che osservano [5] (i `listeners`, nel nostro caso solo `UIDashboard`)

```
public interface LogListener {
    void updateLog(String message);
}

public class ServerImpl extends UnicastRemoteObject implements
↪   ServerInterface {
    //...
    private final List<LogListener> listeners = new
    ↪   LinkedList<>();
    //...
    public void addListener(LogListener toAdd){
        listeners.add(toAdd);
    }
    //...
    public void notifyUpdate(String message){
        for(LogListener l : listeners){
            l.updateLog(message);
        }
    }
}
```

Quando viene chiamato un metodo dell'oggetto remoto da un client viene quindi chiamato il metodo `notifyUpdate()` passando per parametro i dettagli dell'operazione appena effettuata

```
public class UIDashboard extends JFrame implements LogListener,
 ↪  ActionListener {
    //...
    public UIDashboard(){
        //...
        ServerCV.addListener(this);
    }
    //...
    @Override
    public void updateLog(String message) {
        logTextArea.append(message + "\n");
    }
}
```

7 Limiti della soluzione sviluppata

Il principale limite della soluzione sviluppata è il numero massimo di centri vaccinali e di vaccinazioni che è possibile registrare. Questo limite è dettato dalla dimensione dell'ID che deve essere di 16 cifre, e che quindi non ci permette di rappresentare infinite vaccinazioni.
Nello specifico, i primi 5 bit dell'ID identificano il centro vaccinale dove è stata effettuata la vaccinazione, mentre i rimanenti 11 identificano il cittadino vaccinato. Facendo un rapido calcolo arriviamo alla conclusione che è possibile registrare fino a un massimo di 10^5 = 100000 centri vaccinali che, seppur non sia un numero infinito, è comunque largamente sufficiente (anche supponendo ci sia un centro vaccinale per ogni comune d'italia ne avremmo 7904, numero largamente inferiore al limite di 100000). [3]
Discorso analogo per i cittadini, che possono essere massimo 10^{11} = 100000000000, e anche in questo caso possiamo stare tranquilli dato che in italia ci sono circa 60000000 persone, e nel caso in cui tutte si vaccinino, avremmo occupato solo lo 0.06% degli id disponibili!

References

[1] *Stackoverflow*, Online: https://stackoverflow.com/questions/322715/when-to-use-linkedlist-over-arraylist-in-java

[2] *Crypto Stackexchange*, Online: https://crypto.stackexchange.com/questions/67448/what-is-the-time-complexity-of-computing-a-cryptographic-hash-function-random-or

[3] *Wikipedia*, Online: https://it.wikipedia.org/wiki/Italia

[4] *Ionos Digital Guide*, Online: https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/cose-il-singleton-pattern/

[5] *Italian Coders*, Online: https://italiancoders.it/observer-pattern/