VaxCenter

Manuale Tecnico

Versione 1.0

Università degli Studi dell'Insubria – Laurea Triennale in Informatica
Progetto Laboratorio B: VaxCenter

Sviluppato da:
Alessandro Cassani**, matricola **744512
Paolo Bruscagin**, matricola **744703
Damiano Ficara**, matricola **744958
Luca Perfetti**, matricola **746581

Sommario

Introduzione

VaxCenter è un progetto relativo al corso di laboratorio B della facoltà di scienze informatiche dell'Università degli Studi dell'Insubria, svolto durante l'anno accademico 2021-2022. Il progetto è sviluppato in Java 16, e si è utilizzato come tool di gestione di progetto **Maven**. Inoltre, si è utilizzato Java Swing per la realizzazione della UI. il progetto è stato sviluppato su Windows 10, Windows 11 e testato su Windows 10/11, Ubuntu Linux 14.x e Mac OS X 10.8.3+.

Si tratta di un'applicazione stand-alone, la quale non necessita di alcun tipo di installazione specifica nel sistema operativo.

Per maggiori informazioni, si consiglia la visione del manuale utente circa le modalità di installazione di Maven e di Java

Maven e Swing

Alla base del nostro progetto, come detto in precedenza, vi è un tool di gestione di progetto: **"Maven"**. Maven è fondamentale in quanto permette di gestire efficientemente il progetto e ne aumenta la comprensione. Nel progetto sviluppato è stato di fondamentale importanza per l'utilizzo dei plugin (JavaDoc e JAR). Si è scelto di avere nel nostro progetto un POM principale che contenesse i 3 moduli da realizzare nel progetto, affinchè si creasse una relazione `parent` tra il POM principale e i POM dei moduli .

Java Swing è stata scelta per realizzare le interfacce grafiche dell'applicativo.

Librerie esterne utilizzate

Il progetto realizzato utilizza prevalentemente librerie interne, le quali sono necessarie per mettere in pratica i concetti richiesti nelle specifiche. Tuttavia, è stato necessario utilizzare delle librerie esterne per la gestione di componenti grafici più complessi oppure non presenti direttamente nella swing, come ad esempio le animazioni.
Per poter ricercare queste librerie si è utilizzato https://mvnrepository.com/ , un vero e proprio aggregatore di tutte le librerie utilizzabili in maven online.

- **`maven-jar-plugin:`** libreria necessaria per le operazione di build e di creazione del file jar;
- **`postgresql:`** dipendenza necessaria per potersi interfacciare con il database di postgreSQL
- **`maven-javadoc-plugin:`** dipendenza necessaria per la generazione in fase di build della javaDoc
- **`maven-shade-plugin:`** dipendenza necessaria per compiere l'operazione di *package* comprendendo nel *.jar* tutte le dipendenze
- **`timingframework`**: dipendenza che permette di mostrare delle animazioni relative ai bottoni
- **`swingx-all :`** dipendenza che permette di realizzare componenti grafici avanzati

Premesse

- Il numero identificativo non dipende dal singolo centro vaccinale, ma è calcolato sull'insieme degli utenti che si registrano in tutti i centri vaccinali.
- Non può esistere un identificativo uguale per due utenti diversi registrati in due diversi centri vaccinali;
- É stata utilizzata la porta **5432** per il database e la porta **1099** per RMI, in maniera da uniformarci alle porte standard utilizzate;
- Si è utilizzato un database di comuni che rappresenta tutti i comuni presenti in Italia, con rispettiva provincia, Cap e regione, aggiornato al `30/10/2017` : https://rosariociaglia.altervista.org/database-formato-csv-excel-mysql-dei-comuni-italiani-e-citta-scaricare-free-download/

Struttura del programma

Il progetto si articola in 3 moduli:

- **`clientCV:`** contiene le classi relative all'interfaccia grafica con cui l'utente interagisce con l'applicazione. Esso contiene i seguenti package:
  - `centrivaccinali` : contiene le classi con cui si interfaccia l'operatore vaccinale;
  - `cittadini`: contiene le classi con cui si interfaccia utente cittadino;
  - `checkdata`: contiene le classi relative ai controlli per il corretto inserimento dei dati;
  - `graphics`: contiene le classi relative ai componenti grafici personalizzati utilizzati nell'interfaccia grafica;
- **`serverCV:`** contiene le classi relative alla gestione e il funzionamento del server e del relativo database. Esso contiene i seguenti package:
  - `database`: contiene le classi relative all'implementazione dei metodi forniti dal server e alla gestione e connessione del database PostgresSQL.
    - `ui` : contiene le classi relative ai componenti grafici personalizzati utilizzati nell'interfaccia grafica del server;
- **`Util`**: contiene le classi relative alla modellazione degli oggetti gestiti da entrambi i moduli precedentemente elencati;

Il punto di avvio del programma risiede nel modulo:

```
clientCV/centrivaccinali/CentriVaccinali.java
```

Mentre il punto di avvio dell'applicazione server risiede nel modulo:

```
serverCV/database/ServerCentriVaccinali.java
```

Si nota che la suddivisione dell'applicazione nei tre diversi moduli e dei rispettivi package, è stata realizzata al fine di organizzare le classi costituenti il progetto per funzionalità comuni in maniera tale da aumentare la modularità e organizzazione logica del progetto. Il tutto è rappresentato tramite il seguente `packageDiagram`

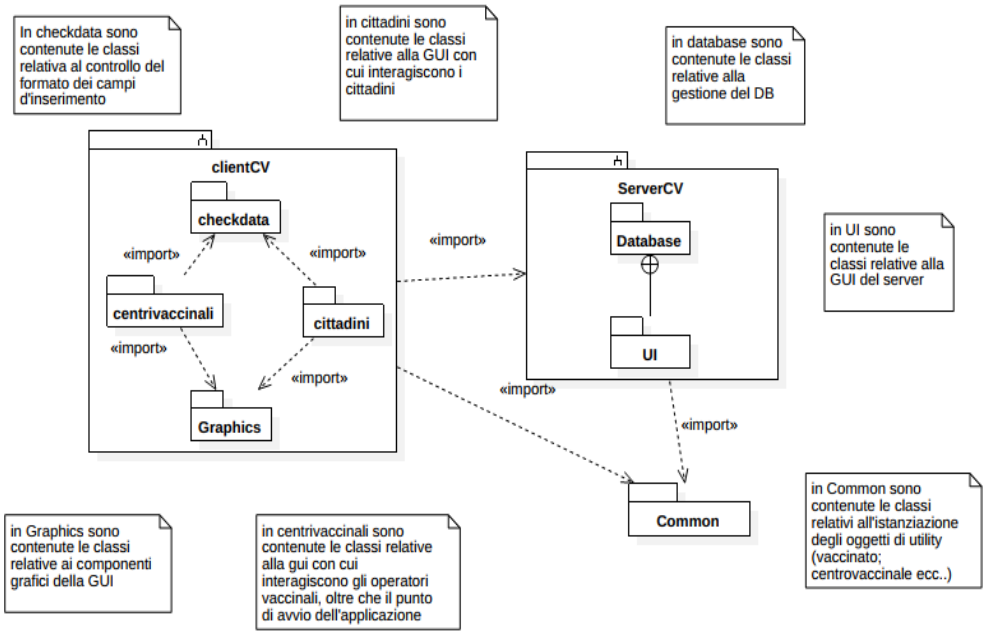

Figura 1: PackageDiagram

Processo di sviluppo

Conoscendo la solidità dei requisiti e la relativa immutabilità, si è scelto di adottare il **modello waterfall** per il ciclo di vita del software.

La prima fase è stata di analisi dei requisiti, seguita da una progettazione esaustiva delle entità e delle interazione fra di esse(UML). Si è poi passati alla fase di implementazione del codice e relativo testing.

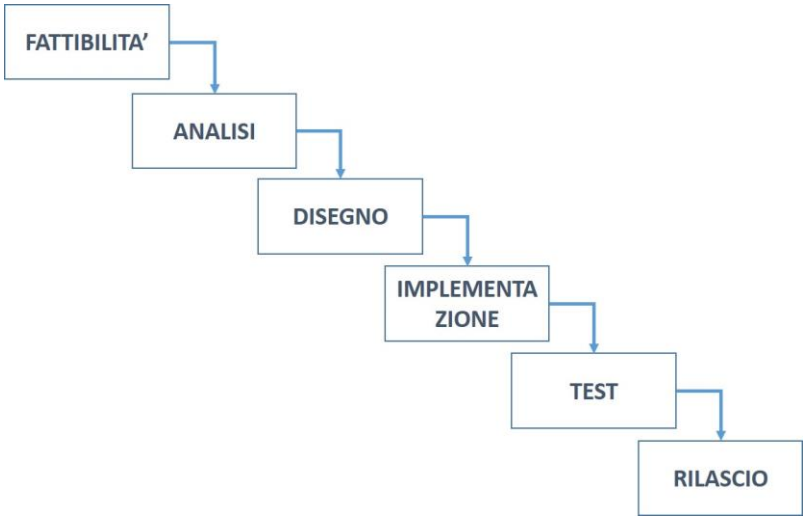

Figura 2: Waterfall

Analisi dei requisiti

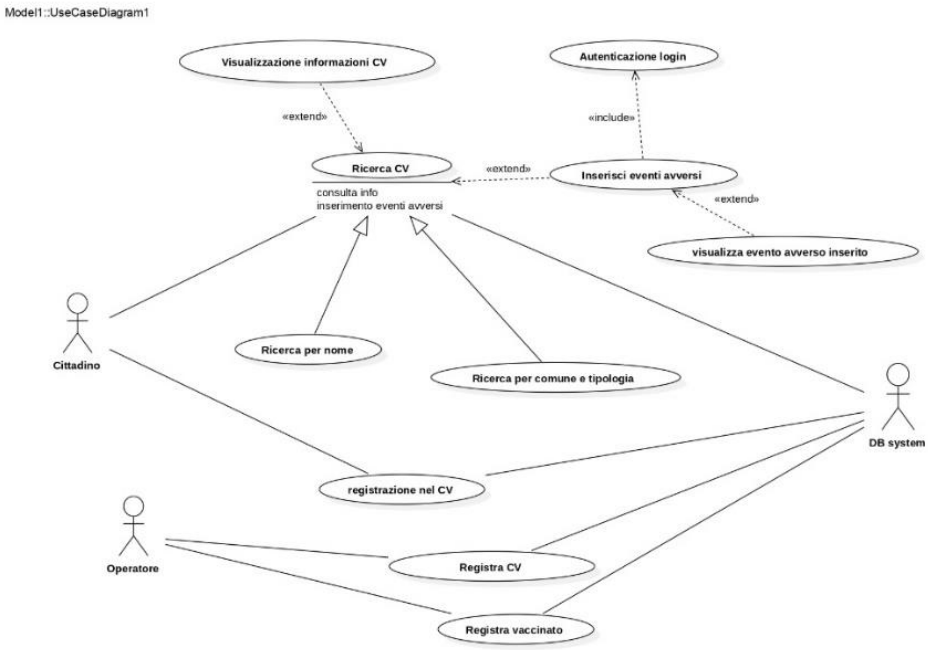

Figura 3 UseCaseDiagram

Analizzando le specifiche si è evidenziata la presenza di 3 attori principali:

- Cittadino
- Operatore
- DB System

i 3 attori realizzano i requisiti dell'applicazione e si è scelto di rappresentare certe funzionalità come estensioni di altre per evidenziare la stretta correlazione e similitudine fra di esse.

Progettazione

Il sistema `client` è strutturato con il seguente `ClassDiagram`

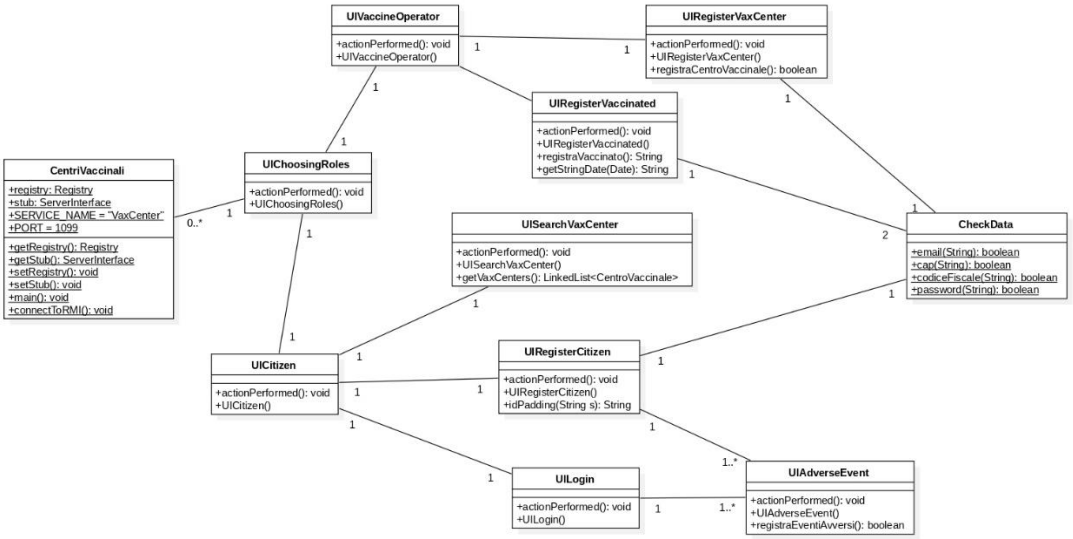

Figura 4 ClientClassDiagram

Di seguito si riporta insieme degli stati(`StateDiagram`) in cui l'utente interagendo con l'applicazione client può trovarsi

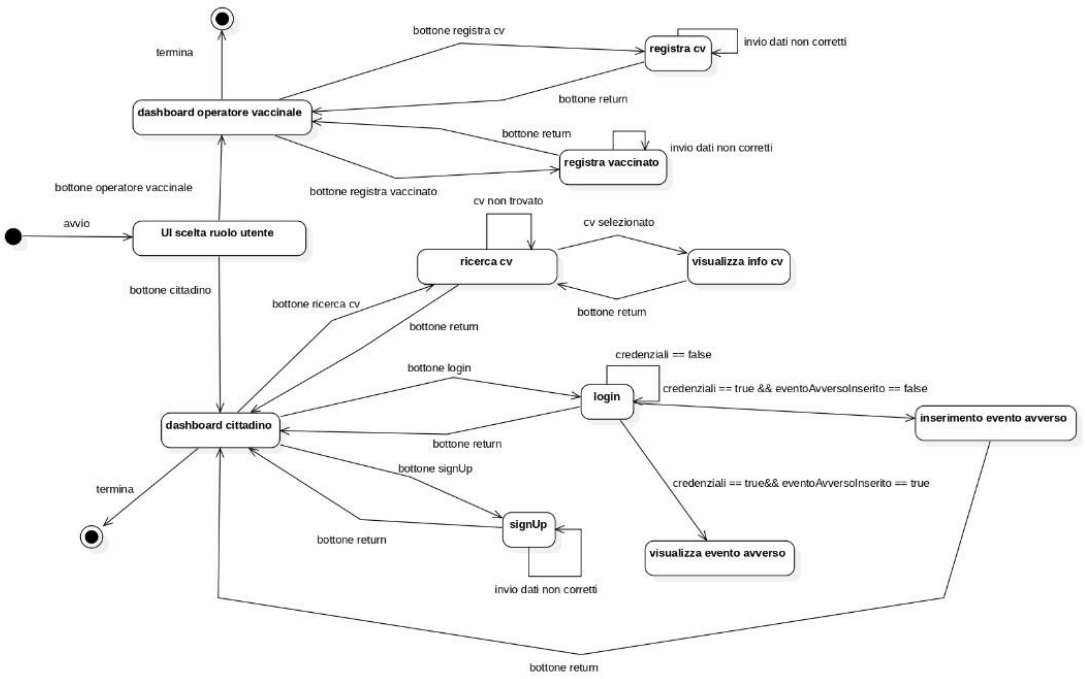

Figura 5 ClientStateDiagram

Il sistema `server` è strutturato con il seguente `ClassDiagram`

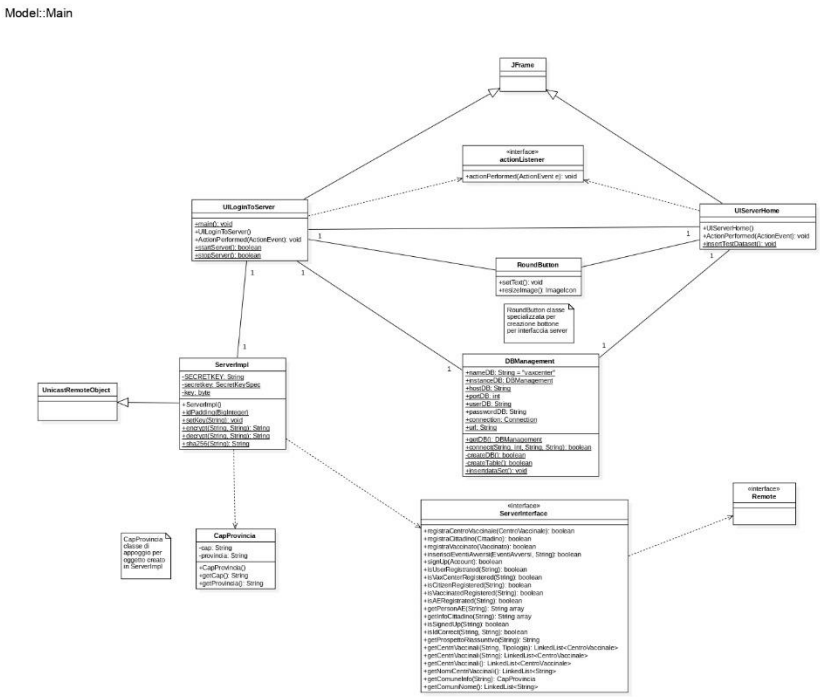

Figura 6 ServerClassDiagram

Di seguito si riporta insieme degli stati(`StateDiagram`) in cui l'utente tecnico interagendo con l'applicazione server può trovarsi.

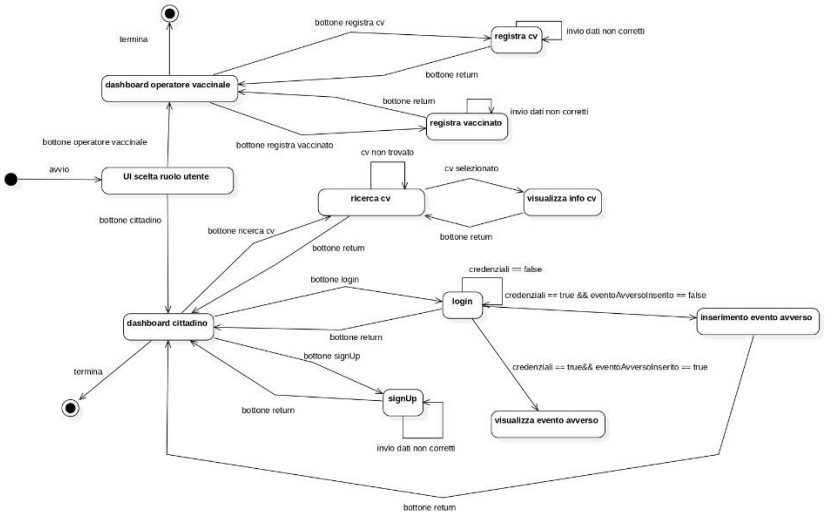

Figura 7 ServerStateDiagram

Il sistema `util` è strutturato con il seguente `ClassDiagram`

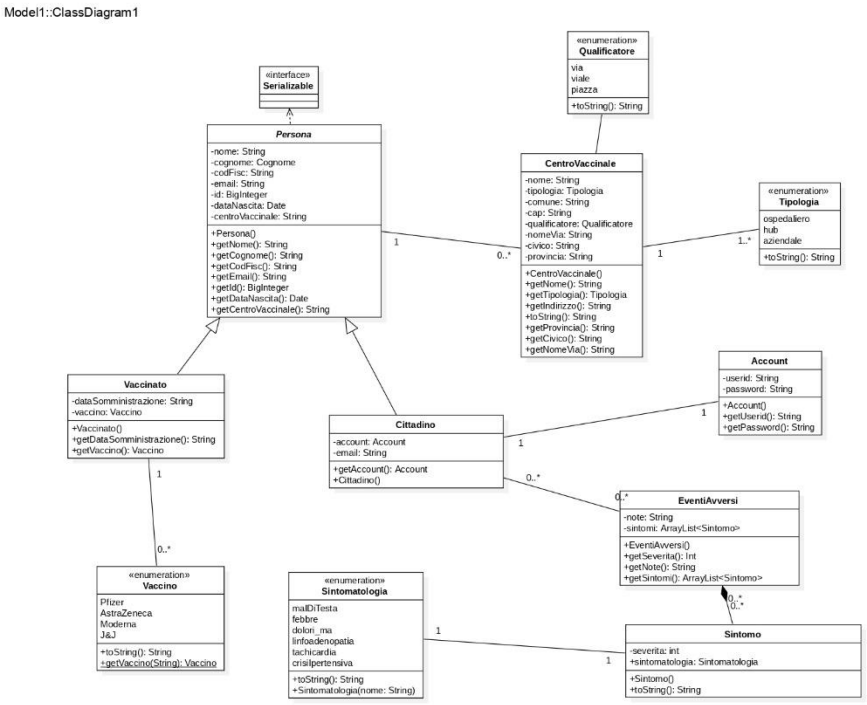

Figura 8 UtilClassDiagram

Particolare importanza nell'applicazione avranno le operazioni di `Login()` e `SignUp()` e `RegistraCentroVaccinale()` , di seguito per una corretta interpretazione e analisi dei possibili scenari, in fase di progettazione si sono realizzati dei `SequenceDiagram` relativi

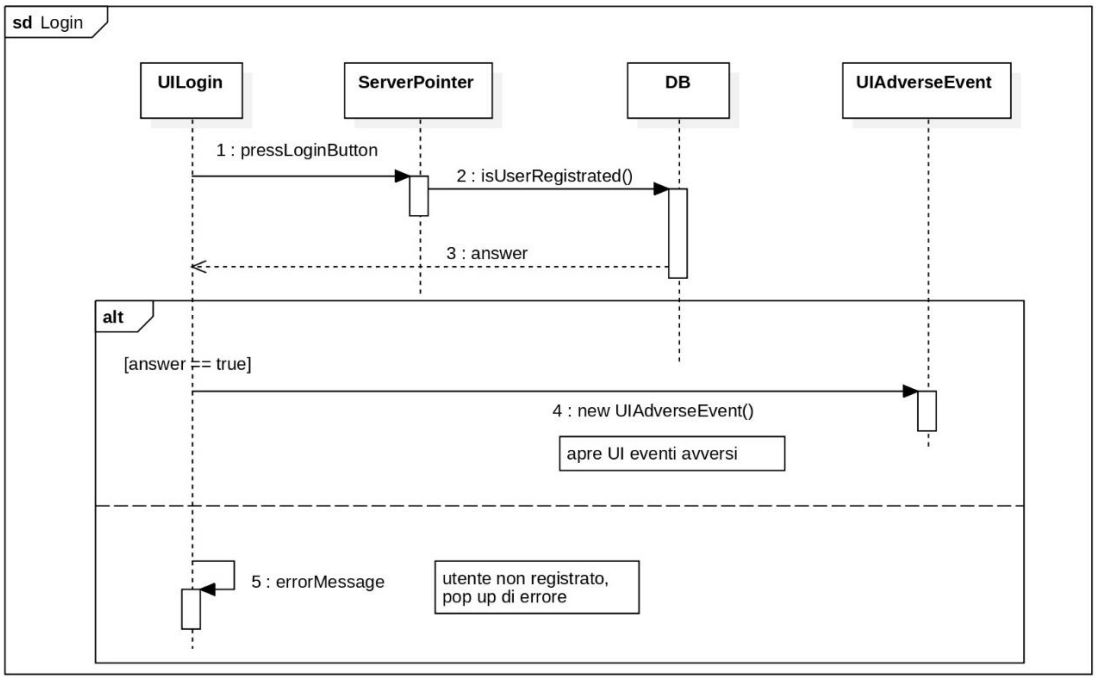

Figura 9 SequenceDiagram Login

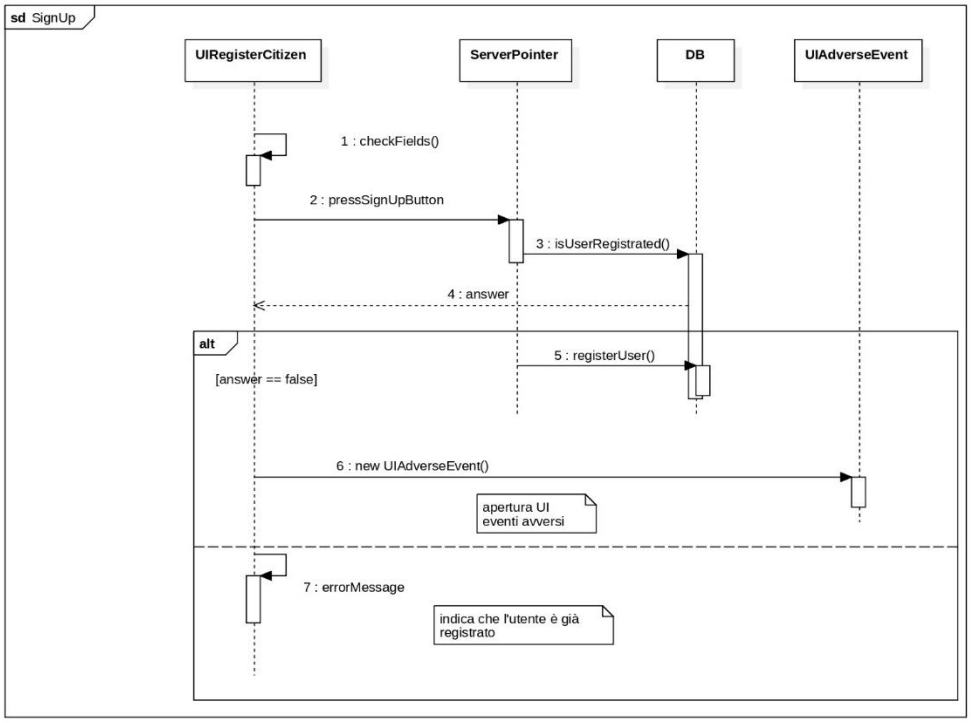

Figura 10 SequenceDiagram SignUp

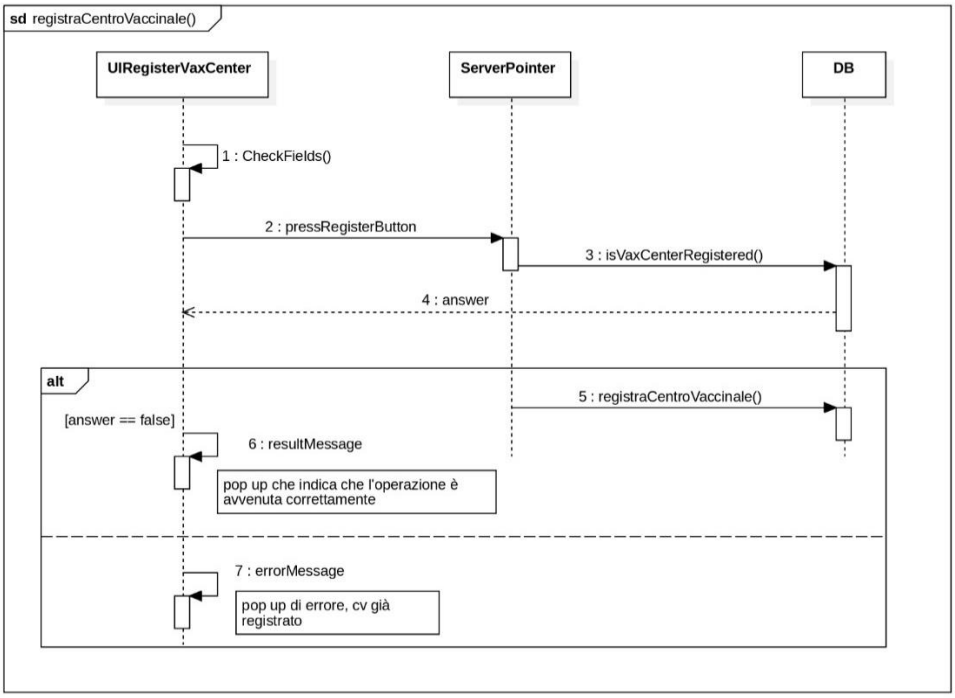

Figura 11 SequenceDiagram registraCentroVaccinale

Progettazione della base di dati

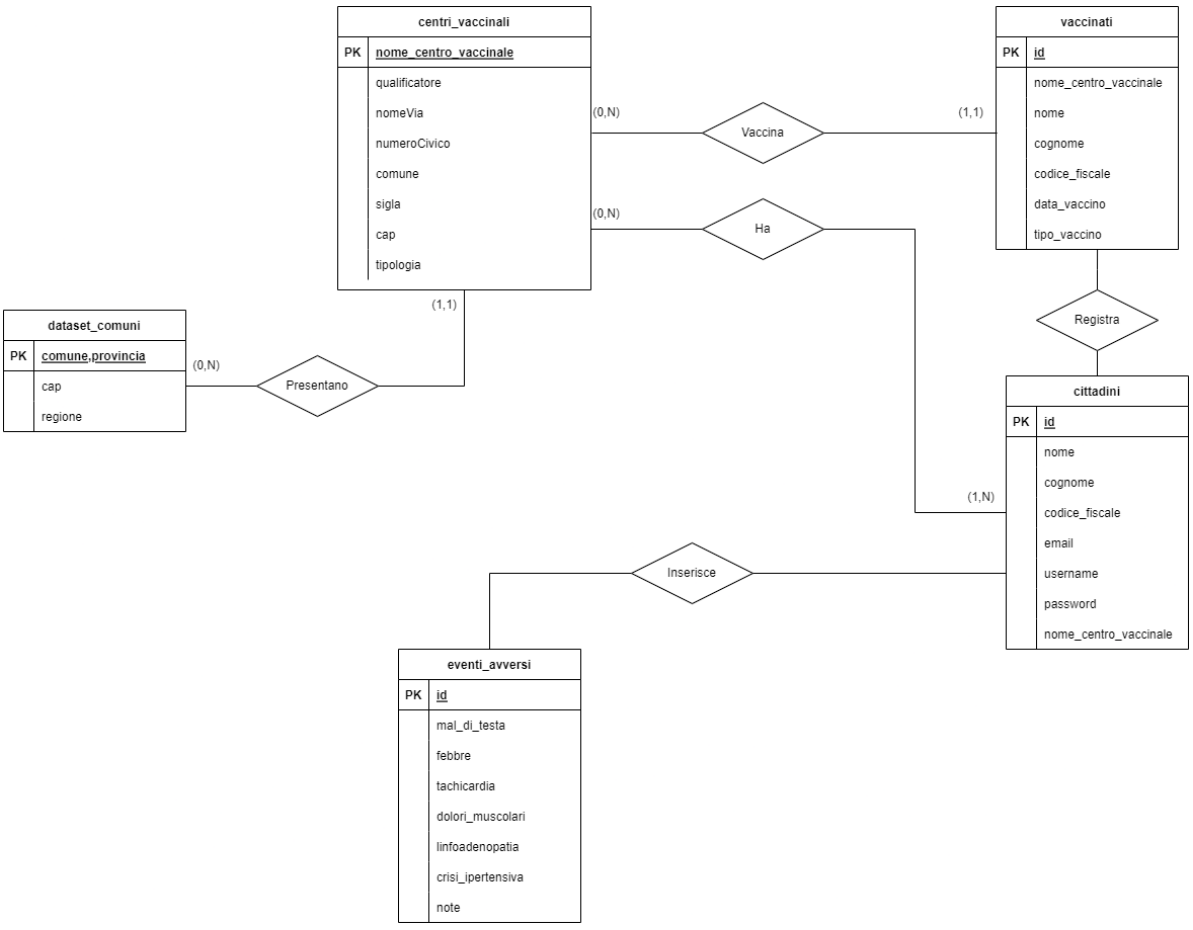

Figura 12 Schema ER

Per la realizzazione dello schema ER si è pensato di associare una tabella ad ogni entità gestita dal sistema come i `Centri Vaccinali, Cittadini, Vaccinati.` Si è resa necessaria l'aggiunta di due ulteriori tabelle, una rappresentante gli `Eventi Avversi` e una contenente l'elenco di tutti i comuni presenti in Italia denominata `dataset_Comuni.` In ognuna di queste relazioni è stata scelta una specifica chiave primaria come **`l'id,`** in quanto univoco per ogni cittadino/vaccinato, oppure come `nome_centro_vaccinale,` poiché si suppone che ogni centro vaccinale abbia un nome univoco. Infine, per la tabella delle località italiane, si è scelto di avere una chiave primaria composta dai campi `comune e provincia`, poiché in Italia ci sono comuni aventi lo stesso nome ma in province differenti.

Si riporta di seguito la traduzione del seguente schema.

Per maggiori dettagli si consiglia di leggere la `Relazione della base di dati` che fornisce un approfondimento chiaro ed esaustivo.

```
CREATE TABLE vaccinati(
    id VARCHAR(16) PRIMARY KEY,
    nome_centro_vaccinale VARCHAR(80) REFERENCES centri_vaccinali,
    nome VARCHAR(50),
    cognome VARCHAR(50),
    codice_fiscale VARCHAR(50),
    data_vaccino VARCHAR(40),
    tipo_vaccino VARCHAR(50)
);


CREATE TABLE centri_vaccinali(
    nome_centro_vaccinale VARCHAR(50) PRIMARY KEY,
    qualificatore VARCHAR(7),
    nome_via VARCHAR(30),
    civico VARCHAR(6),
    provincia CHAR(2),
    comune VARCHAR(30),
    cap INTEGER,
    tipologia VARCHAR(20)
    FOREIGN KEY(comune,provincia) REFERENCES dataset_comuni
);
```

```
CREATE TABLE cittadini(
    id VARCHAR(16) PRIMARY KEY REFERENCES vaccinati,
    nome VARCHAR(50),
    cognome VARCHAR(50),
    codice_fiscale VARCHAR(50),
    email VARCHAR(50),
    username VARCHAR(50),
    password VARCHAR(150),
    nome_centro_vaccinale VARCHAR(50) REFERENCES centri_vaccinali
);
CREATE TABLE eventi_avversi(
    id VARCHAR(16) PRIMARY KEY REFERENCES cittadini,
    mal_di_testa INTEGER,
    febbre INTEGER,
    tachicardia INTEGER,
    dolori_muscolari INTEGER,
    linfoadenopatia INTEGER,
    crisi_ipertensiva INTEGER,
    note VARCHAR(256)
);
CREATE TABLE dataset_comuni(
    comune VARCHAR(40),
    provincia VARCHAR(2),
    cap INTEGER,
    regione VARCHAR(21),
    PRIMARY KEY(comune,provincia)
);
```

Classi e Funzionalità principali

Si presentano in seguito l'implementazione delle classi contenenti le funzionalità principali richieste, per maggiori informazioni si consiglia la visione della **javaDoc**.

clientCV

Nei prossimi paragrafi si tratteranno le classi principali del modulo `clientCV`.

ServePointer

La classe `ServerPointer` è la classe che permette la comunicazione del client con il server remoto. I metodi di questo modulo sono tutti statici in maniera tale da poter richiamare sempre il riferimento del server e utilizzare i metodi implementati da esso. Particolare rilevanza viene data al metodo `connectToRMI()` che, come dice il nome stesso, permette di memorizzare le informazioni necessarie alla comunicazione RMI. Per la ricerca dell'oggetto server nel registry, caricato sulla macchina server, si è utilizzato un metodo della classe `Registry`, ossia il metodo `lookup()`.

Si è deciso, a seguito della buona riuscita dell'operazione, di proiettare una schermata di benvenuto (`WelcomeScreen`) per segnalare l'avvio dell'applicativo e rendere l'applicazione comprensibile e intuitiva.

```
   public static void connectToRMI(){

        host = hostName.getText().toString();

        try {

             ServerPointer.setRegistry(LocateRegistry.getRegistry(host, PORT));
             ServerPointer.setStub((ServerInterface)
             ServerPointer.getRegistry().lookup(SERVICE_NAME));

        }catch (RemoteException e) {

            JOptionPane.showMessageDialog(null, " SERVER OFFLINE !",
"Messaggio",JOptionPane.ERROR_MESSAGE);

            System.exit(0);

        } catch(NotBoundException e) {

            JOptionPane.showMessageDialog(null, " SERVIZIO NON TROVATO !",
"Messaggio",JOptionPane.ERROR_MESSAGE);

            System.exit(0); }}
```

Metodo che gestisce la connessione ad RMI

Si nota come vengano gestite tutte le possibili casistiche avvalendoci di **pop-up** grafici. La presenza di un errore di tipo `SERVER OFFLINE` è dovuta al fatto che, se il server non è in funzione, non è possibile utilizzare i metodi ad esso associati. Un errore di `SERVIZIO NON TROVATO` si verifica quando il servizio caricato nel registry della macchina server non viene trovato.

Classi UI

Le classi `UI` sono tutte le classi che rappresentano le varie finestre dell'applicazione e tutte le possibili operazioni che si possono compiere. Si è deciso, in fase di progettazione, di utilizzare una convezione interna, ossia di porre tutte le classi con il prefisso UI seguito dal tipo di interfaccia grafica che viene implementata. Si è valutato di porre in ogni costruttore di queste classi le istruzioni relative alla parte grafica, come ad esempio la dimensione della finestra oppure il colore dello sfondo, in maniera tale che, quando si crea l'oggetto di tale classe, verrà creata l'interfaccia relativa.

Quasi la totalità di queste classi implementa un metodo `actionPerformed()`, che permette di gestire gli eventi associati al `listener` dei componenti grafici, oppure si ha l'utilizzo di metodi `MouseEvent()`, che, come si evince dal nome, permettono di gestire gli eventi collegati al cursore del sistema operativo.

Per poter utilizzare i metodi del server , definiti nella classe `ServerImpl`, si preleva il riferimento del server(avvalendoci dei metodi statici della classe `ServerPointer)` e poi si richiamano i metodi di registrazione, come ad esempio `registraCittadino()`. Per rendere il tutto più comprensibile e intuitivo si è deciso di definire un metodo privato interno alla classe il quale, a partire dai componenti grafici, salvi i valori e li associ ad un oggetto(in questo caso *cittadino)* . Infine, mediante i metodi del server , si memorizza nel database.

```
private boolean registraCittadino(){

        String nomeCentro = Objects.requireNonNull(nomeCV.getSelectedItem()).toString();

        String name = nomeCittadino.getText().toUpperCase();

        String surname = cognomeCittadino.getText().toUpperCase();

        String cf = codiceFiscale.getText().toUpperCase();

        String mail = email.getText();

        String userid = userID.getText();

        String ID = IDUnivoco.getText();

        String pwd = password.getText();

        try {

            ServerPointer.getStub().registraCittadino(new Cittadino(

                    name,surname,cf,mail,new BigInteger(ID),nomeCentro,new
Account(userid,pwd)));

            return true;

        } catch (RemoteException | SQLException ex) {

            return false;

        } }
```

Da sottolineare inoltre l'utilizzo di immagini che, per comodità, in fase di sviluppo sono state poste in una cartella `resources`, per richiamarle basterà seguire il percorso nel progetto e utilizzare il nome associato ad esse .

UILogin

La classe `UILogin` è la classe che permette al cittadino di compiere il login, tale classe utilizza dei campi grafici appositamente realizzati ossia `RoundButton, MyPwdField e MyTextField.` Inoltre, prevede un numero massivo di tentavi di prova e in caso di esaurimento di essi il sistema si arresterà . Tale decisione è stata presa per implementare delle policy di sicurezza ed evitare attacchi di forza bruta.

UISearchVaxCenter

La classe `UISearchVaxCenter` permette di ricercare i centri vaccinali. Per rendere intuitiva e intelligente la ricerca si è utilizzata una tabella che si aggiornasse dinamicamente a seconda dell'input del cittadino, per far questo è stato necessario implementare dei campi grafici apposti come `SearchField, InfoSearch` e `InfoSearchEvent.`

Componenti grafiche

Nell'applicazione realizzata è stato necessario creare dei componenti grafici aggiuntivi , questo per poter rendere la grafica moderna ed estremamente accessibile e comprensibile a qualsiasi utente la utilizzi. Per maggiori dettagli si consiglia di consultare la **javaDoc** , dove si ha un quadro completo e chiaro di quello che è stato realizzato.

Controlli

Al fine di garantire integrità dei dati e correttezza di essi, si è deciso di realizzare dei controlli che controllino se e-mail, codice fiscale, e ID seguano un **pattern** specifico. Ad esempio, la mail deve presentare un @ all'interno di essa, oppure il codice fiscale deve essere sintatticamente corretto e seguire determinate regole ed infine ID che deve avere una lunghezza specifica.

serverCV

Nei prossimi paragrafi si tratteranno le classi principali del modulo `serverCV`.

ServerImpl

La classe ServerImpl contiene l'implementazione dei metodi relativi alle funzionalità rese disponibili dal server. In questo modulo sono presenti anche i metodi utilizzati per garantire la riservatezza dei dati sensibili degli utenti. Questa classe implementa l'interfaccia `ServerInterface` contenente la segnatura dei metodi del server ed estende `UnicastRemoteObject`, operazioni necessarie per la realizzazione di un sistema distribuito tramite l'API java `RMI`.

Essa contiene inoltre la definizione di tre campi statici:

- `SECRETKEY`: costante di tipo String rappresentante la chiave segreta con cui vengono cifrate le informazioni sensibili degli utenti
- `SecretKey`: campo statico di tipo `SecretKeySpec`, contiene la chiave generata da un array di byte per il nostro algoritmo di cifratura (**AES**).
- `Key`: contiene un array di byte usato in fase di generazione della chiave segreta.

Si è scelto di procedere con la cifratura delle sole informazioni sensibili tramite l'algoritmo di cifratura **AES**, con la modalità **ECB**. Le informazioni in chiaro attraversano un processo di padding gestito dall'algoritmo **PKCS5**.

Si è scelta la modalità **ECB** (**Electronic CodeBook**) perché si ha la necessità che, data la stessa chiave, il cipherText generato dall'algoritmo sia sempre uguale dato lo stesso PlainText. Questa caratteristica si è resa necessaria per l'implementazione dei controlli usati nell'applicativo. La debolezza più significativa a livello di sicurezza è che la chiave è scritta in chiaro nel sorgente del codice.

```
public static String encrypt(final String strToEncrypt, final String secret) {
        try {
            setKey(secret);
            Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
            return Base64.getEncoder()
                  .encodeToString(cipher.doFinal(strToEncrypt.getBytes("UTF-8")));
        } catch (Exception e) {
            System.out.println("Error while encrypting: " + e.toString());
        }
        return null;
    }
```

Metodo di cifratura

Si è fatta eccezione per la cifratura delle password, esse non sono cifrate con **AES** ma bensì con la **funzione di hash one-way SHA-256**. Si è fatta la seguente decisione in base all'irreversibilità di questa funzione, essendo che dal *cipherText* non è possibile ricostruire il testo in chiaro e così scoprire la password inserita. Nell'implementazione di SHA-256 viene utilizzato un *messageDigest*, ovvero una funzione di hash unidirezionale sicura che accetta dati di dimensioni arbitrarie e restituiscono un valore hash di lunghezza fissa. I dati da codificare sono formattati secondo la codifica **UTF-8** che assegna ad ogni carattere Unicode una sequenza specifica di bit.

```
public static String sha256(final String base) {
        try{
            final MessageDigest digest = MessageDigest.getInstance("SHA-256");
            final byte[] hash = digest.digest(base.getBytes("UTF-8"));
            final StringBuilder hexString = new StringBuilder();
            for (int i = 0; i < hash.length; i++) {
                final String hex = Integer.toHexString(0xff & hash[i]);
                if(hex.length() == 1)
                    hexString.append('0');
                hexString.append(hex);
            }
```

```
            return hexString.toString();
        } catch(Exception ignored){}
        return base;
}}
```

Algoritmo sha256

I metodi per effettuare le registrazioni delle informazioni inserite sono i seguenti:

1. `Boolean registraCentroVaccinale()`
2. `Boolean registraCittadino()`
3. `String registraVaccinato()`
4. `Boolean inserisciEventiAvversi()`

Questi metodi ritornano un oggetto di tipo `Booleano` (eccetto `registraVaccinato()`) indicando così l'esito dell'operazione e segnalando nel caso la presenza di eccezioni sollevate. Viene sempre applicato il modificatore `synchronized`, onde evitare race conditions nel caso di due inserimenti contemporanei. Prima di ogni registrazione viene ricercato il riferimento al DB e alla relativa connessione (operazione eseguita all'avvio del programma), successivamente si invoca su di esso il metodo `preparedStatement(String sql)`. Quest'ultimo permette la creazione di un oggetto di tipo `Prepare` alla query inserita come stringa. In seguito, viene eseguito l'update della base di dati e chiuso il `preparedStatement` con il metodo `close()`.

```
public synchronized boolean registraCentroVaccinale(CentroVaccinale centroVaccinale)
throws RemoteException{
        try {
            Connection con = DBManagement.getDB().connection;
            PreparedStatement ps = con.prepareStatement("INSERT INTO
centri_vaccinali(nome_centro_vaccinale,qualificatore,nome_via,civico,provincia,comune,cap
,tipologia) "
                    + "VALUES (?,?,?,?,?,?,?,?)");
            ps.setString(1, centroVaccinale.getNome());
            ps.setString(2,centroVaccinale.getQualificatore().toString());
            ps.setString(3,centroVaccinale.getNomeVia());
            ps.setString(4,centroVaccinale.getCivico());
            ps.setString(5,centroVaccinale.getProvincia());
            ps.setString(6,centroVaccinale.getComune());
            ps.setInt(7,centroVaccinale.getCap());
            ps.setString(8,centroVaccinale.getTipologia().toString());
            ps.executeUpdate();
            ps.close();
        } catch(SQLException e){ e.printStackTrace();return false;}
        return true;
    }
```

Metodo per registrare un centro vaccinale

Particolare attenzione viene data al metodo registraVaccinato(), perché oltre all'inserimento delle informazioni di un cittadino post-vaccinazione, in esso è contenuto anche la logica per la generazione dell'ID (identificativo univoco).

Esso è salvato come `BigInteger`.

Per prima cosa viene eseguita una query che permette di avere l'insieme di tutti gli ID registrati in ogni centro vaccinale, successivamente questo risultato viene inserito in un `TreeSet`, il quale ha la caratteristica di ordinare di default il suo contenuto in ordine crescente. In seguito, si estrae l'ultimo elemento della struttura dati tramite il metodo last(), che rappresenta quindi l'elemento maggiore. Da questo valore si genera il nuovo ID del vaccinato sommando 1 al valore estratto.

Ora si hanno tutte le informazioni necessarie alla registrazione e quindi si procede in maniera analoga descritta al passo precedente.

Ultima differenza è che, come presentato in precedenza, questo è l'unico metodo che non ritorna un booleano ma una stringa. Questo perché essendo l'identificativo generato all'interno del metodo, si vuole avere un modo per renderlo visibile al client che procede alla registrazione. Questo metodo, a differenza degli altri, ha complessità in tempo e spazio O(n), perché per ogni inserimento è necessario ricreare il TreeSet inserendo tutti i vaccinati.

Questa caratteristica degrada le prestazioni del software man mano che il numero di vaccinati registrati cresce. Per poter comunque segnalare la presenza di errori in fase di esecuzione si è scelto di ritornare **la stringa "-1"** per gestire le eccezioni.

```
public synchronized String registraVaccinato(Vaccinato vaccinato) throws RemoteException
{
        BigInteger numero;
        try {
            PreparedStatement    preparedStatement   =
             DBManagement.getDB().connection.prepareStatement("SELECT   id     FROM
             vaccinati");
            ResultSet resultSet = preparedStatement.executeQuery();
            TreeSet<BigInteger> id = new TreeSet<>();
            while(resultSet.next()){
                id.add(new BigInteger(resultSet.getString(1)));
            }
            if(!id.isEmpty())
                numero = id.last().add(new BigInteger("0000000000000001"));
            else
                numero = new BigInteger("0000000000000000");
            preparedStatement.close();
             PreparedStatement   ps=
             DBManagement.getDB().connection.prepareStatement("INSERT   INTO
            ps.setString(1, idPadding(numero));
            ps.setString(2,vaccinato.getCentroVaccinale());
            ps.setString(3,encrypt(vaccinato.getNome(),SECRETKEY));
            ps.setString(4,encrypt(vaccinato.getCognome(),SECRETKEY));
            ps.setString(5,encrypt(vaccinato.getCodFisc(),SECRETKEY));
            ps.setString(6,encrypt(vaccinato.getDataSomministrazione(),SECRETKEY));
            ps.setString(7,encrypt(vaccinato.getVaccino().toString(),SECRETKEY));
            ps.executeUpdate();
            ps.close();

        }catch (SQLException e){e.printStackTrace();return "-1";}
        return idPadding(numero);
    }
```

Metodo per registrare un vaccinato

I seguenti metodi rappresentano le interrogazioni al DB necessarie per implementare i controlli negli inserimenti:

1. `boolean isAEregistrated()`
2. `boolean isSignedUp()`
3. `boolean isUserRegistrated()`
4. `boolean isVaxCenterRegistrated()`
5. `boolean isVaccinatedRegistrated()`

6. `boolean isIdCorrect()`

Ognuno di questi metodi ritorna un `booleano`, il quale risultato permette di conoscere se l'inserimento di una specifica informazione rispetta ed è coerente con le regole dell'applicazione.

Non è stato inserito il modificatore `synchronized` perché eseguendo delle "letture" non **possono crearsi situazioni di race conditions.**

Per prima cosa viene sempre ricercato il riferimento al DB, poi in maniera analoga agli inserimenti viene creato un oggetto di tipo `PreparedStatement` passando come parametro una semplice query per selezionare gli elementi della base di dati che sono utili ai nostri controlli. In seguito, il risultato della query viene inserito in un oggetto di tipo `ResultSet` e tramite un `if` contenente la condizione `resultSet.next()` si viene a conoscenza della effettiva presenza dei dati. Per come sono pensati e utilizzati i controlli, essi danno esito negativo (ovvero bloccano l'operazione che l'utente è intenzionato a fare) se l'interrogazione al DB porta anche solo ad un singolo dato. È per questo che, se il metodo `resultSet.next() == true`, il metodo ritorna false essendoci un dato già inserito (essendo che `resultSet.next()` passa al prossimo elemento presente nel `ResultSet` e quindi al primo elemento).

```
public boolean isVaccinatedRegistrated(String cf) throws RemoteException {
        try {
            PreparedStatement ps = DBManagement.getDB().connection.prepareStatement("SELECT * FROM vaccinati WHERE codice_fiscale = ?");
            ps.setString(1, encrypt(cf,SECRETKEY));
            ResultSet resultSet = ps.executeQuery();
            if(resultSet.next()){
                return true;
            }
        }catch (SQLException e){
            return false;
        }
        return false;
    }
```

Metodo per verificare se un cittadino si vaccinato o meno

I metodi che permettono la ricerca di un insieme di oggetti nel DB e li rendono disponibili ai client sono i seguenti:

1. `String[] getPersonAE()`: ritorna gli eventi avversi inseriti da un cittadino
2. `String getID()`: ritorna l'ID di un cittadino
3. `String[] getInfoCittadino()`: ritorna le informazioni di un cittadino

4. `String[] getProspettoRiassuntivo()`: ritorna il prospetto riassuntivo inserito da un cittadino
5. `LinkedList<> getCentriVaccinali(String comune, Tipologia tipologia)`: ritorna un insieme di centri vaccinali scelti per comune e tipologia
6. `LinkedList<> getCentriVaccinali(String nome)`: ritorna un insieme di centri vaccinali scelti per nome
7. `LinkedList<> getComuneInfo()`: ritorna CAP e provincia di un comune
8. `LinkedList<> getComuniNome()`: ritorna l'insieme di comuni presenti

La procedura è analoga all'implementazione dei metodi dei controlli, con la sola differenza che quando si itera il `ResultSet` con il ciclo `while` si ricostruisce ad ogni iterazione l'oggetto prelevato dal DB e lo si inserisce in una struttura dati adeguata.

Non è stato inserito il modificatore `synchronized` perché eseguendo delle "letture" **non possono crearsi situazioni di race conditions.**

I metodi che ritornano un array sono metodi che di default ritornano un insieme di valori di dimensione conosciuta, per questo si è scelto di utilizzare una struttura dati statica. I metodi che ritornano una `LinkedList` sono metodi che, al contrario dei precedenti, racchiudono un insieme di valori di dimensione non nota ma variabile a runtime e per questo necessitano di una struttura dati dinamica.

È stato scelto di utilizzare una `LinkedList` al posto di `un'ArrayList` perché, nonostante entrambe abbiano complessità in termini di spazio e tempo O(n), la `LinkedList` utilizza al suo interno una doppia `LinkedList` che agevola la manipolazione dei dati ed evita di applicare degli shift ai bit rendendola anche più performante.

```
public String[] getInfoCittadino(String id) throws RemoteException {

        PreparedStatement preparedStatement = null;

        String [] info = new String [6];

        try {

            preparedStatement  =  DBManagement.getDB().connection.prepareStatement("SELECT
* FROM cittadini WHERE id = ?");

            preparedStatement.setString(1,idPadding(new BigInteger(id)));

            ResultSet resultSet = preparedStatement.executeQuery();

            while(resultSet.next()){

                info[0] = resultSet.getString(1);

                info[1] = decrypt(resultSet.getString(2),SECRETKEY);

                info[2] = decrypt(resultSet.getString(3),SECRETKEY);

                info[3] = decrypt(resultSet.getString(4),SECRETKEY);

                info[4] = decrypt(resultSet.getString(5),SECRETKEY);

                info[5] = decrypt(resultSet.getString(6),SECRETKEY);

            }
```

```
            return info;
        } catch (SQLException e) {
            return null;
        }
    }
```

Metodo che permette di ottenere le informazioni di base di un cittadino

DBManagement

La classe `DBManagement` ha lo scopo di gestire la creazione del database **PostgreSQL** e la gestione della sua connessione. Essa contiene i metodi per la creazione delle tabelle delle entità gestite dall'applicativo.

Questo modulo contiene i campi statici:

- `String nameDb`
- `DBManagement istanceDB`
- `String hostDb`
- `Int portDB`
- `String userDb`
- `String passwordDb`
- `Connection connection`
- `String url`

Utili per il salvataggio dei dati di accesso al DB.

Pattern Singleton

Questa classe utilizza il Pattern **Singleton** per la creazione di un oggetto. Questa scelta è data dal fatto che dall'avvio dell'applicazione lato server, è necessario creare il collegamento al DB una sola volta all'inizio. È quindi più conveniente salvare il riferimento e riutilizzarlo piuttosto che ricreare la connessione ogni volta ad ogni interazione di ogni client con la base di dati (peggiorerebbe di gran lunga le prestazioni).

All'avvio del server si memorizza all'interno di **`instanceDB`** (che è inizializzato a NULL) il riferimento al database appena creato con le informazioni inserite tramite UI e assegnate ai corrispettivi campi statici tramite il metodo **`connect`**(). Alla prossima interazione di un client con il `server instanceDB` non sarà NULL e si avrà accesso al riferimento del DB da lì.

```
public static DBManagement getDB(){

        if(instanceDB == null) {

            instanceDB = new DBManagement();

           connect(UILoginToServer.getHostTextField(),
UILoginToServer.getPortTextField(), UILoginToServer.getUserTextField(),

                    UILoginToServer.getPswTextField()

           );

        }

        return instanceDB;

    }
```

Singleton

Per una visione più accurata del processo di creazione delle tabelle e dei relativi vincoli si rimanda alla relazione della base di dati.

Il metodo `connect()` si occupa quindi di instaurare la vera e propria connessione al DB, tramite il metodo `DriverManager.getConnection(String url)`. Se la connessione non è presente `(connection = null)` allora il database non è ancora stato creato e viene quindi chiamato il metodo `createTable()` per la creazione delle tabelle. In seguito, è presente la chiamata al metodo `insertDataset()`, per l'inserimento del dataset dei comuni province e CAP italiani. Se invece la connessione è già stata effettuata viene ripresa accedendo al campo statico connection.

```
connection = DriverManager.getConnection(url + nameDB, userDB, passwordDB);
```

Metodo per la creazione della connessione al DB

Particolare attenzione si vuole porre al metodo per l'inserimento delle informazioni relative alle località italiane. È stato scaricato un DB online in formato `.csv` e successivamente tramite il tool online `tableConvert` (in bibliografia) lo si è trasformato in un file SQL, generando i relativi inserimenti. Questi sono stati inseriti nel progetto sotto forma di file `.txt` nelle risorse del modulo `serverCV`. All'avvio dell'applicazione server per la prima volta, il software in seguito alla creazione delle tabelle legge gli inserimenti dal file ed esegue man mano l'update della tabella **`dataset_comuni`**, registrando così le informazioni desiderate.

Questi inserimenti nella relazione precedente hanno lo scopo di aumentare l'usabilità del sistema, dato che l'utente finale durante l'inserimento dei dati selezionando il nome del comune avrà direttamente in output la provincia e il relativo CAP. Questa scelta porta anche ad una diminuzione del tasso di errori inseriti dall'utente evitando la creazione di dati inconsistenti.

```
private static void insertDataSet() {
        try {
            PreparedStatement ps;
            URL resource = DBManagement.class.getResource("/dataset/dataset_comuni");
            File dataset = null;
            String ds = "";
            assert resource != null;
            dataset = Paths.get(resource.toURI()).toFile();
            assert dataset != null;
            BufferedReader br = new BufferedReader(new FileReader(dataset));
            StringBuilder sb = new StringBuilder();
            String line = br.readLine();
            int i = 0;

            while (line != null) {
                sb.append(line);
                sb.append("\n");
                line = br.readLine();
                i++;
            }
            ds = sb.toString();
            getDB().connection.prepareStatement(ds).executeUpdate();
        }catch(Exception e){e.printStackTrace();}
    }
```

Metodo per l'inserimento dei dati delle località italiane

ServerHome

La classe `ServerHome` è l'unico modulo di interfaccia grafica lato server che verrà presentato. Per una visione accurata delle altri classi si consiglia la visione della relativa javaDoc.

Si è scelta la presentazione di questo componente perché esso rappresenta la dashboard con cui il personale tecnico si accinge all'accensione e allo spegnimento del server. È presente un bottone con cui si ha la possibilità di inserire un dataset di test contenente le informazioni di vaccinati, cittadini e di eventuali eventi avversi riscontrati.

Per la gestione degli eventi della UI, è stato implementato il metodo `actionPerformed()`, associando dei `listener` ai bottoni della grafica.

Le operazioni di accensione e spegnimento del server sono implementate nei metodi statici `startServer()` e `stopServer()`.

Nella procedura di accensione viene creato il `registry` nella porta selezionata da UI. Il `registry` consiste nella locazione in cui verrà depositato l'oggetto `ServerImpl` contenente le funzionalità del server.

```
registry = LocateRegistry.createRegistry(PORT);
```

Metodo per la creazione del registry

Per caricare l'oggetto nel `registry` si utilizza il metodo **`rebind()`** applicato al riferimento del registry, passando come parametri **l'etichetta** con cui sarà riconoscibile l'oggetto dai client e **l'oggetto stesso**.

```
registry.rebind(SERVICE_NAME, server);
```

Metodo per il caricamento dell'oggetto server nel registry

I client, richiedendo i servizi, accederanno alla porta della macchina in cui è caricato il registry e preleveranno da esso l'oggetto `ServerImpl` salvandolo e utilizzandolo come se risiedesse in locale.

Nella procedura di spegnimento l'oggetto ServerImpl viene semplicemente rimosso dal registry tramite il metodo `UnicastRemoteObject.unexportObject(String objectName, boolean force)`, rendendolo non più disponibile ai client.

```
UnicastRemoteObject.unexportObject(server, true);
```

Si è scelto di settare la variabile `boolean force` a `true` in modo che, nel momento in cui viene rimosso l'oggetto server, ogni chiamata pendente ed in corso ai metodi del server venga bloccata a prescindere dal suo esito.

util

Nel modulo **util** si è deciso di porre tutte quelle classe che non effettuano operazioni di calcolo sui dati ma rappresentano gli oggetti reali di interesse , replicando con un'elevata astrazione le loro caratteristiche principali ossia:

- `Account;`
- `CentroVaccinale`
- `Cittadino`
- `EventiAvversi`
- `Persona`
- `Qualificatore`
- `Sintomatologia`
- `Sintomo`
- `Tipologia`
- `Vaccino`
- `Vaccinato`

Da notare utilizzo di `enumerativi`, i quali molto comodi per usare set di constanti correlate, come ad esempio i vaccini, i qualificatore o i possibili sintomi presenti. Particolare attenzione viene posta sulla relazione di ereditarietà presente tra `Cittadino-Persona e Vaccinato-Persona`. Persona è una classe **astratta**, contenente campi comuni sia per Vaccinato che per Cittadino. Si è scelto di porre questa relazione in modo da aumentare la riusabilità del codice ed evitare la presenza di campi privati ridondanti nelle due sottoclassi.

Bibliografia

https://www.geeksforgeeks.org/introduction-apache-maven-build-automation-tool-java-projects/

https://www.javatpoint.com/java-swing

https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/SecretKeySpec.html

https://www.javatpoint.com/difference-between-arraylist-and-linkedlist#:~:text=1)%20ArrayList%20internally%20uses%20a,elements%20are%20shifted%20in%20memory.

https://www.techtarget.com/searchsecurity/definition/Electronic-Code-Book

https://www.herongyang.com/Cryptography/DES-JDK-What-Is-PKCS5Padding.html

https://sectigostore.com/blog/sha-256-algorithm-explained-by-a-cyber-security-consultant/

https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html

https://tableconvert.com/excel-to-sql

https://stackoverflow.com/questions/423950/rounded-swing-jbutton-using-java

https://github.com/DJ-Raven/textfield-search-option/tree/main/src/textfield

https://mvnrepository.com

https://github.com/alessandroCassani/VaxCenter_B

https://www.salute.gov.it/portale/vaccinazioni/homeVaccinazioni.jsp

https://lab24.ilsole24ore.com/numeri-vaccini-italia-mondo/

https://docs.oracle.com/javase/tutorial/uiswing/misc/splashscreen.html

https://docs.oracle.com/javase/tutorial/uiswing/misc/splashscreen.html

https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/