

APP VACCINI



MANUALE
TECNICO

Università degli Studi dell'Insubria
Laurea Triennale in Informatica

Progetto Laboratorio B:
Sviluppato da:

- Ademi Qaldo, matricola 746362
- Sassi Gabriele, matricola 745081
- Brullo Enrico, matricola 744949
- Battaglia Simone, matricola 744514

SOMMARIO

INTRODUZIONE	2
SCELTE PROGETTUALI.....	2
Use-Case Diagram	3
Class Diagram	4
Sequence Diagram	7
State Diagram	13
PROGETTAZIONE DEL DATABASE	14
Schema Concettuale.....	14
Ristrutturazione del risultato.....	15
Schema logico.	15
Vincoli di dominio e integrità	16
STRUTTURA GENERALE DELLE CLASSI	18
Client Lab	19
Server Lab.....	21
STRUTTURE DATI UTILIZZATE	23
SCELTE ALGORITMICHE	25
CONNESSIONE A DATABASE	32
PATTERN UTILIZZATI	35
MVC	35
Singleton.....	36
Proxy	36
Skeleton.....	37
MAVEN.....	38
SITOGRAFIA	41

INTRODUZIONE

“Vacciniamo” è un progetto sviluppato per Laboratorio B per il corso di laurea in Informatica dell'università degli studi dell'Insubria.

Il progetto è sviluppato in Java 16, usa un'interfaccia grafica costruita con Swing ed è stato sviluppato e testato su:

- Window 10-11
- Mac Os Ventura 13.0 -13.0.1
- Linux popOS.

Il funzionamento generale dell'applicazione consiste in un semplice software in grado di gestire la campagna vaccinale COVID-19. In particolare, gli operatori vaccinali possono amministrare il form dedicato all'aggiornamento dello stato delle vaccinazioni. I cittadini possono visualizzare informazioni sull'andamento della campagna e attraverso la propria area personale sono in grado di gestire funzionalità riguardanti la propria vaccinazione.

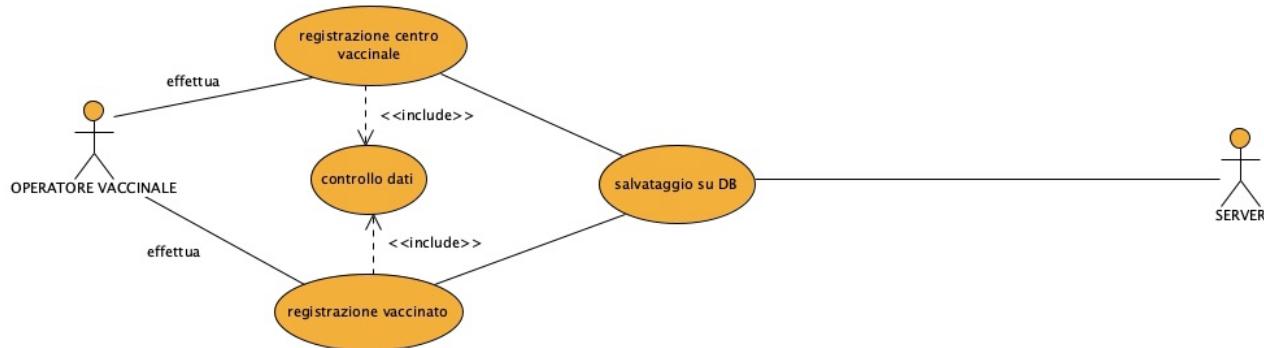
SCELTE PROGETTUALI

Prima del vero e proprio sviluppo del progetto, è necessario introdurre le principali scelte progettuali rappresentati da diagrammi UML: linguaggio comune visivo di modellazione strutturale e comportamentale di applicazioni software. In particolare, i seguenti diagrammi descrivono i confini, la struttura ed il comportamento del sistema e degli oggetti al suo interno:

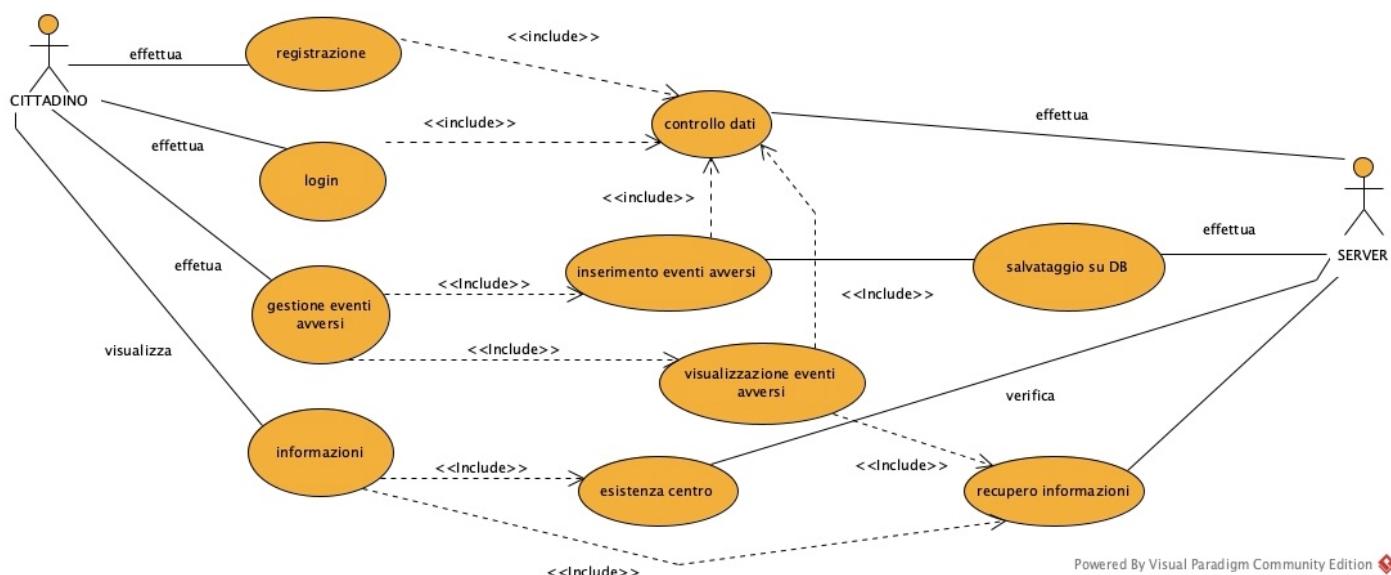
1. Use-Case Diagram.
2. Class Diagram.
3. Sequence Diagram.
4. State Diagram.

- **USE-CASE DIAGRAM** → definisce il comportamento dell'applicazione: tipicamente mostra un'interazione tra il sistema e l'utente, rappresentando i requisiti del sistema, ovvero funzioni offerte ed "identificabili" dall'utente. Uno-use case ha connessioni con uno o più attori, descritte come figure esterne che usufruiscono delle funzionalità sistema, qui rappresentate da:
 - Operatore Vaccinale.
 - Cittadino.
 - Server.

- Operatore Vaccinale: Requisiti sistema



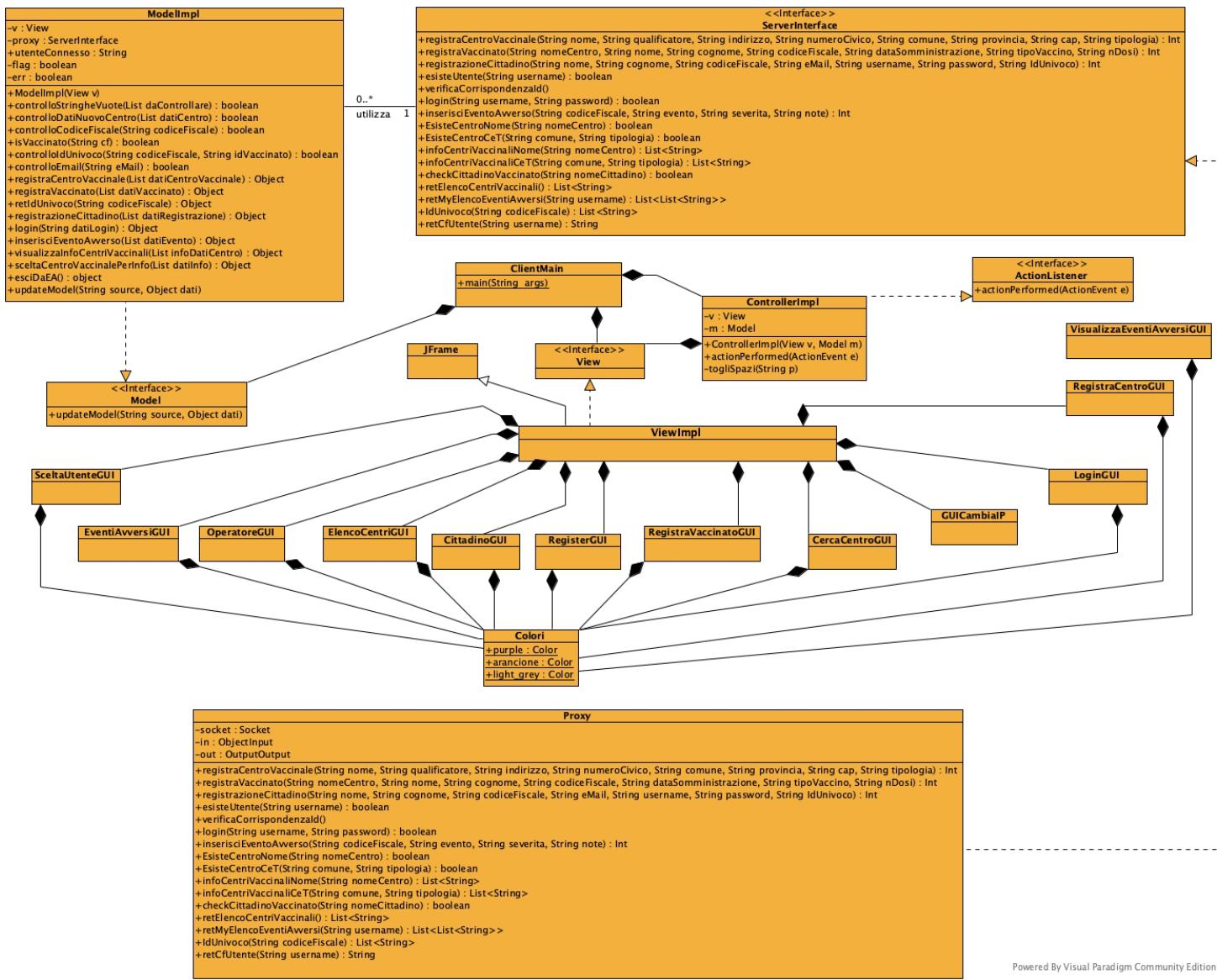
- Cittadino: Requisiti sistema



Powered By Visual Paradigm Community Edition

- **CLASS DIAGRAM** → modello più importante poiché definisce lo scheletro del sistema e gli elementi base, specificando classi e relazioni tra esse.

- ClientLAB: schema generale

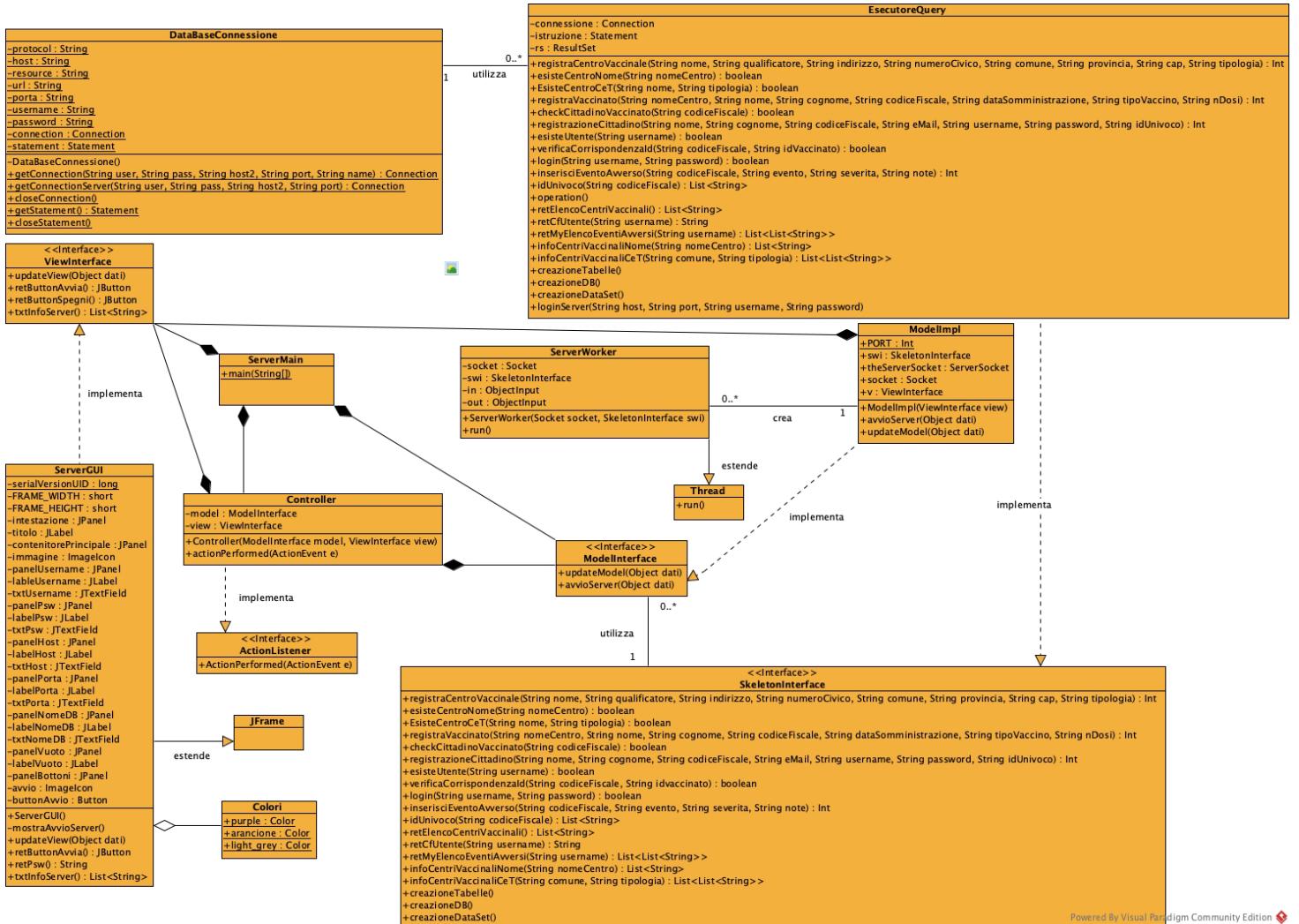


Powered By Visual Paradigm Community Edition

- ClientLAB: View Grafiche



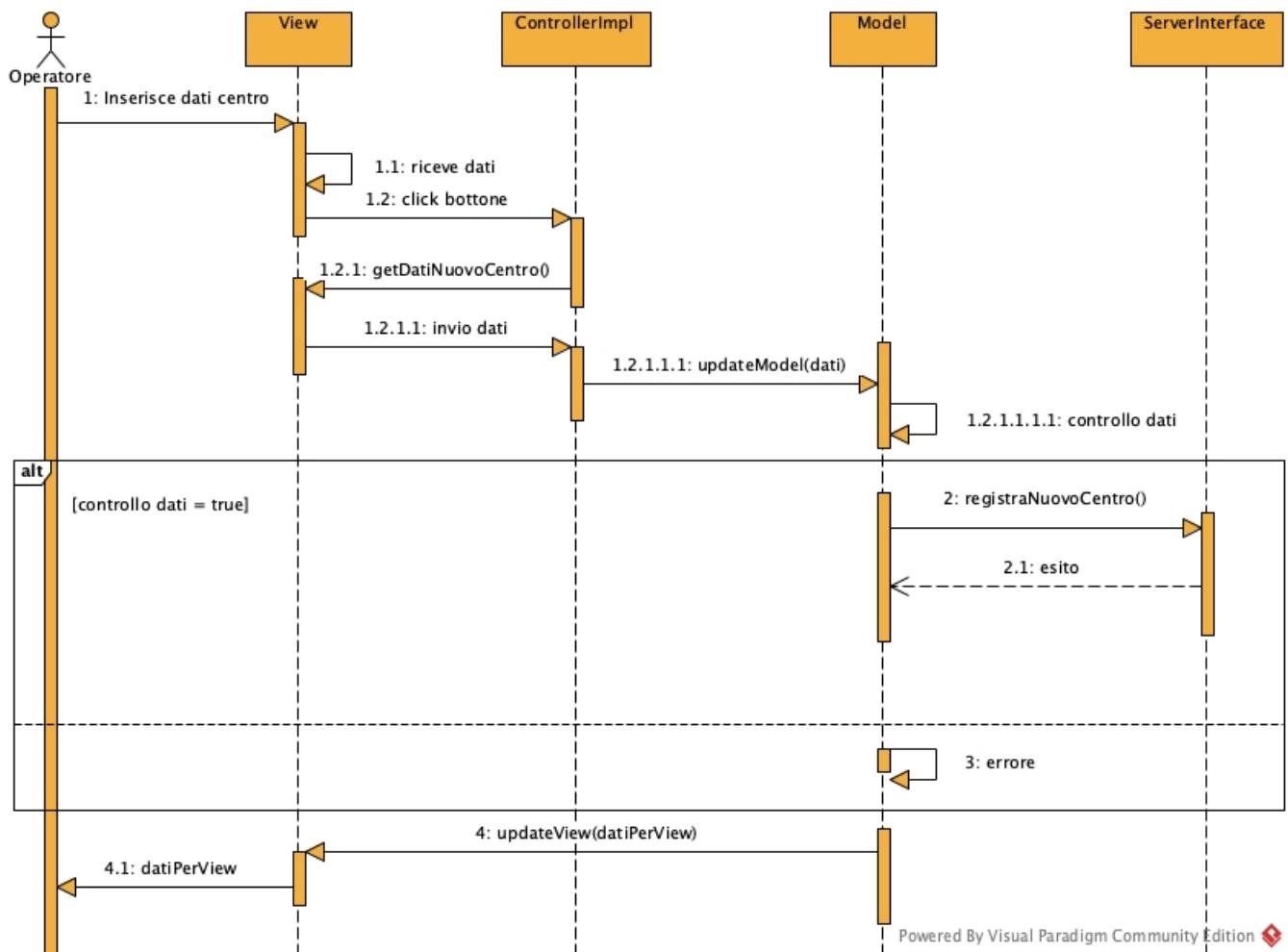
- ServerLAB: Schema Generale



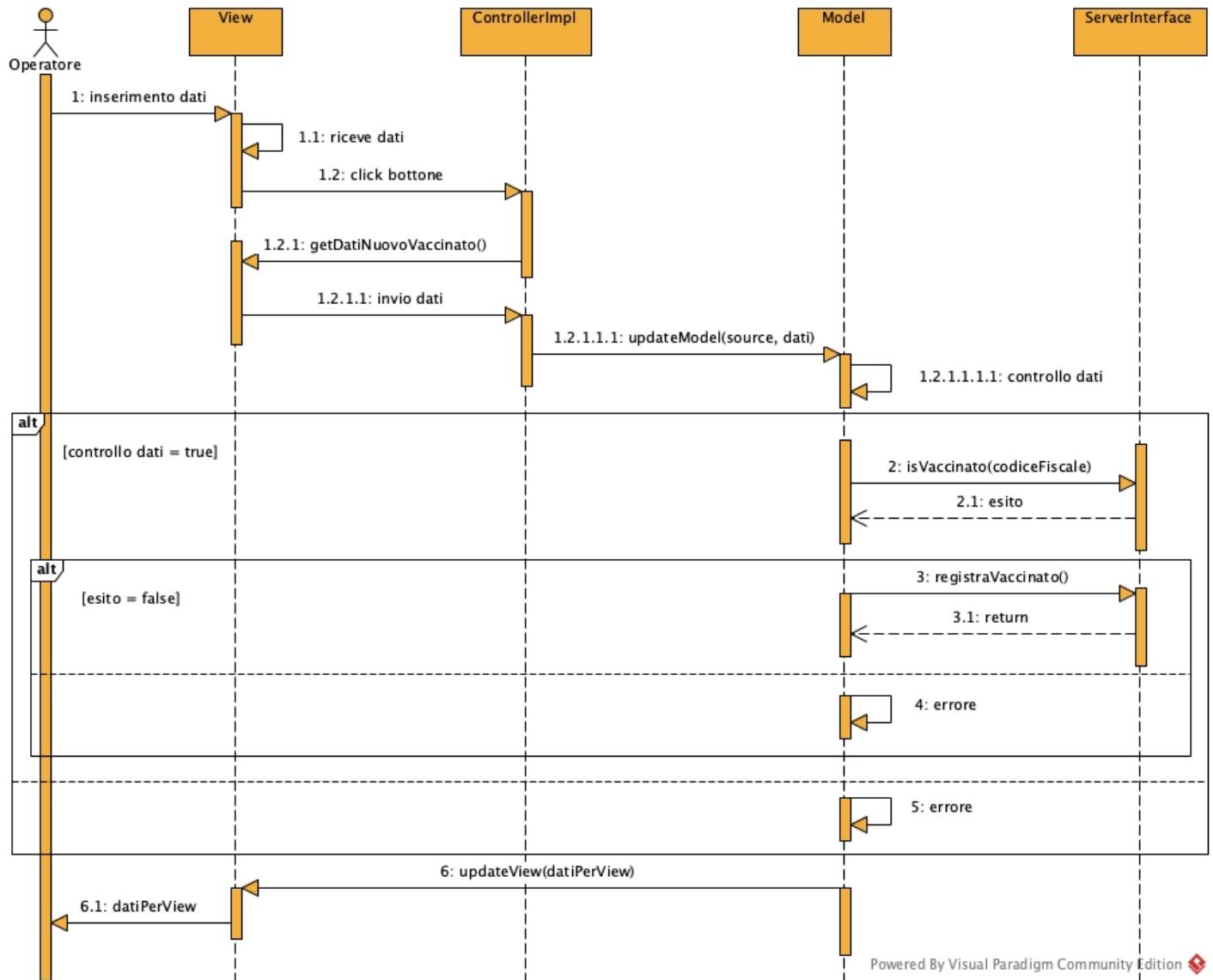
Powered By Visual Paradigm Community Edition

- **SEQUENCE DIAGRAM** → Descrivono il comportamento dinamico di un gruppo di oggetti che interagiscono per risolvere un problema. Rappresentano il comportamento di uno Use-Case o scenario specifico. In particolare, i sequence diagram evidenziano la sequenza temporale delle azioni con oggetti e sequenze di messaggi scambiati.
 - **Operatore Vaccinale** → le funzionalità messe a disposizione degli operatori vaccinali descritte mediante diagrammi sono le seguenti:
 - Registrazione centro vaccinale.
 - Registrazione cittadino vaccinato.

- **Registrazione centro vaccinale:**



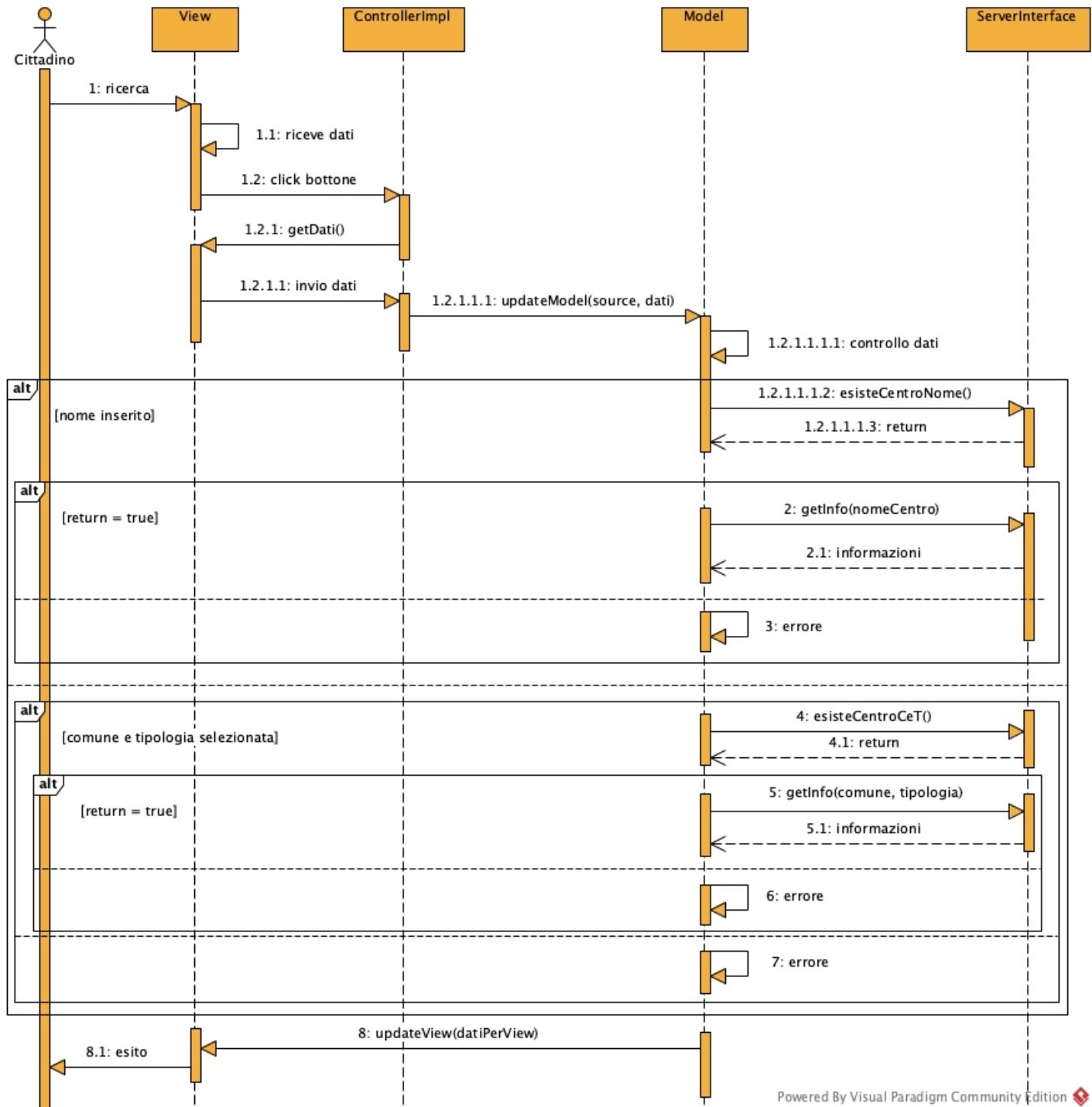
- Registrazione cittadino vaccinato:



Powered By Visual Paradigm Community Edition

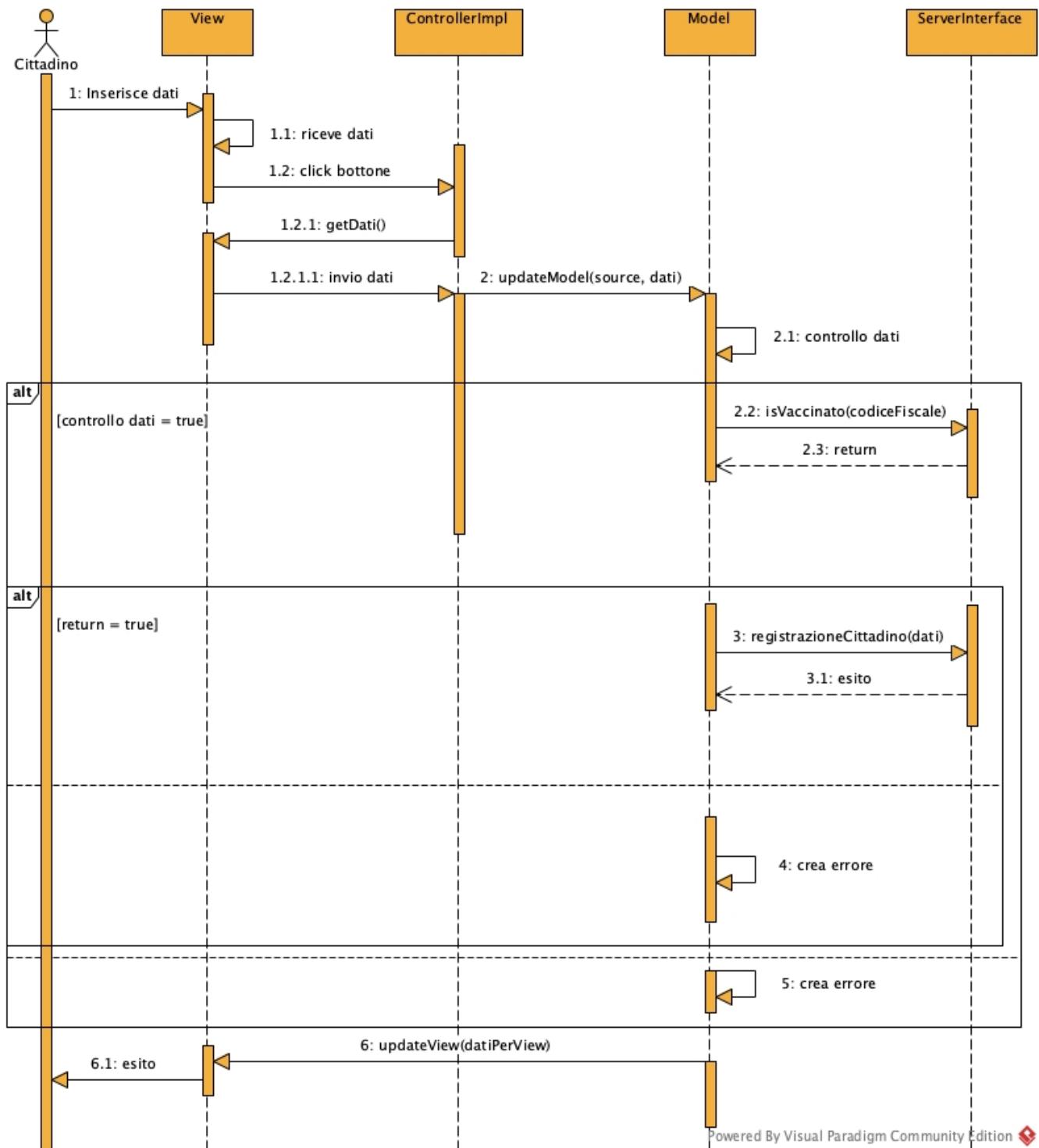
- **Cittadino** → le funzionalità messe a disposizione dei cittadini descritte mediante diagrammi sono le seguenti:
 - Visualizza info centri vaccinali.
 - Registrazione area riservata.
 - Login.
 - Inserimento Eventi avversi.

- Visualizza info centri vaccinali:



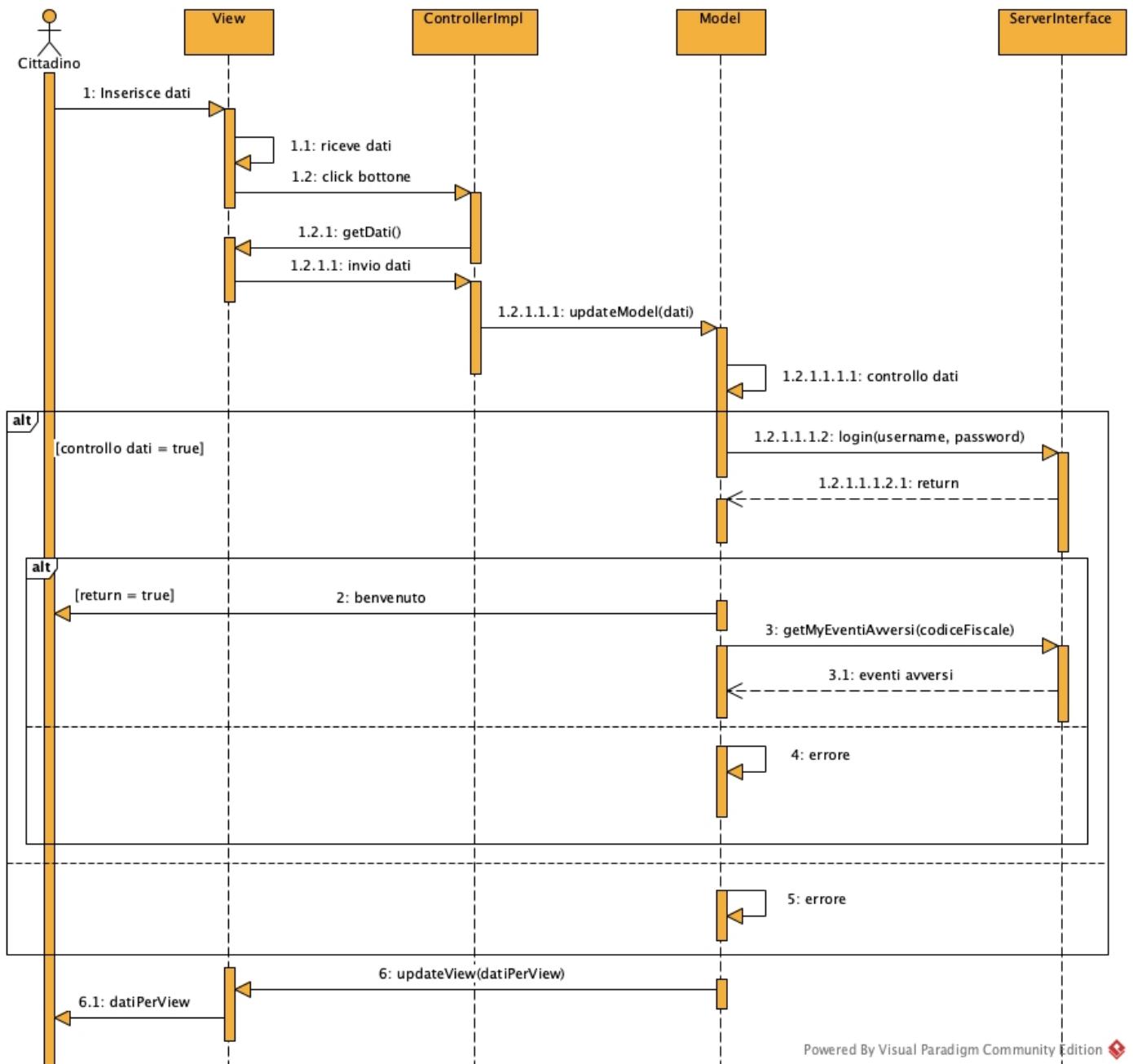
Powered By Visual Paradigm Community Edition

- **Registrazione area riservata:**



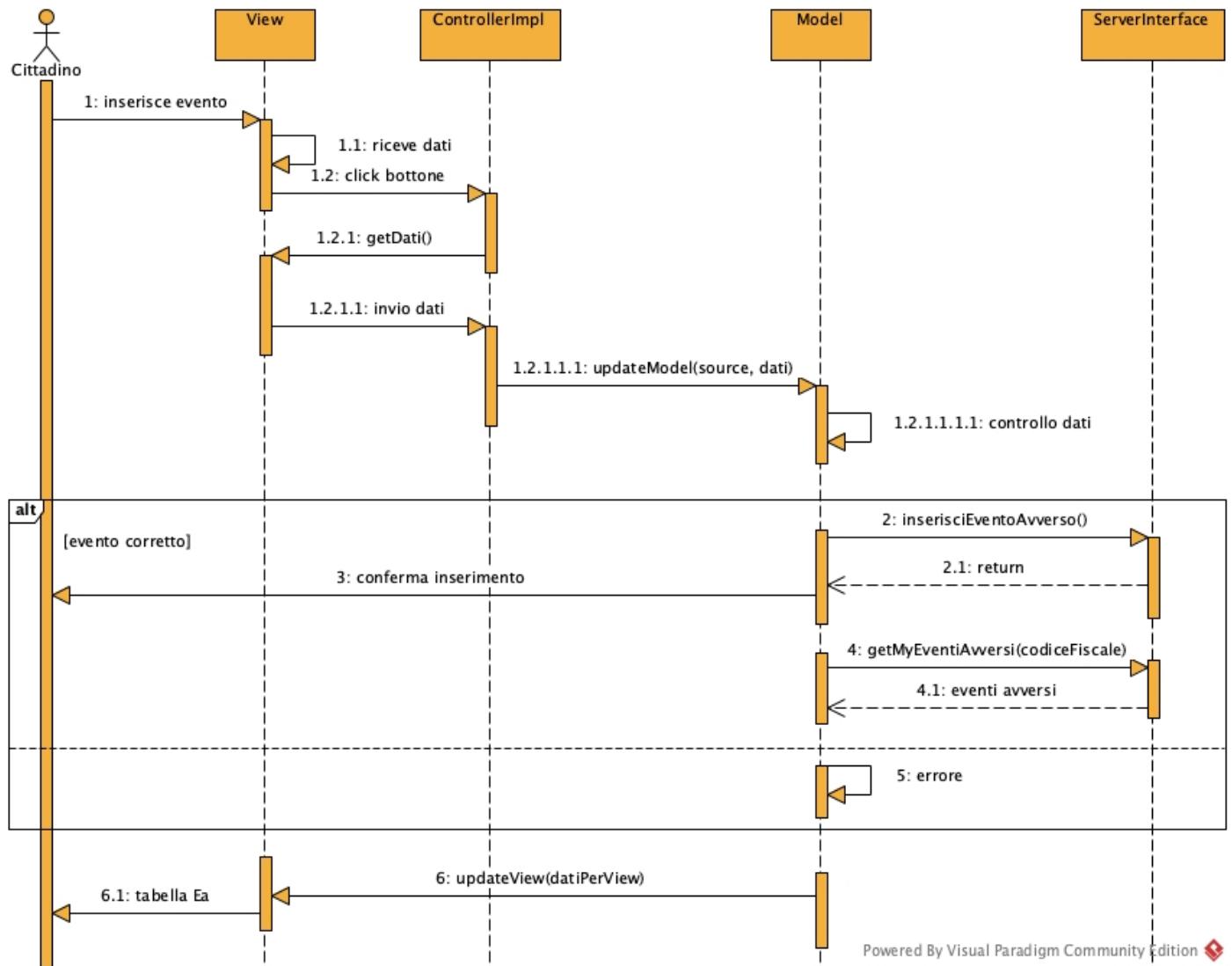
Powered By Visual Paradigm Community Edition

- **Login:**



Powered By Visual Paradigm Community Edition

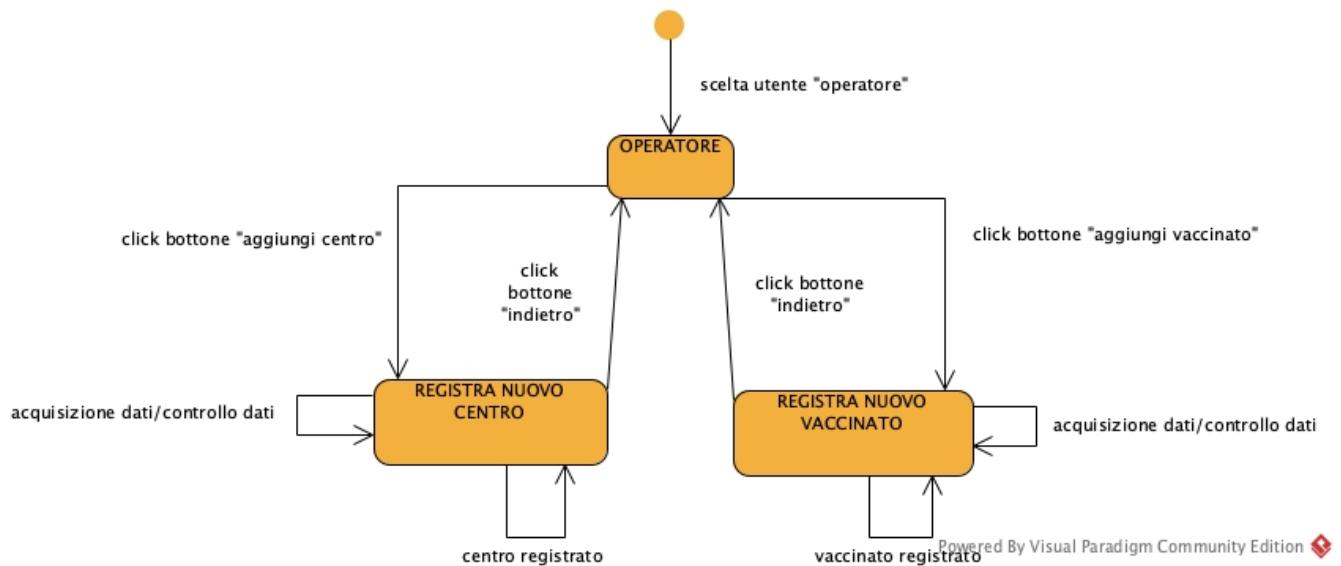
- Inserimento Eventi Avversi:



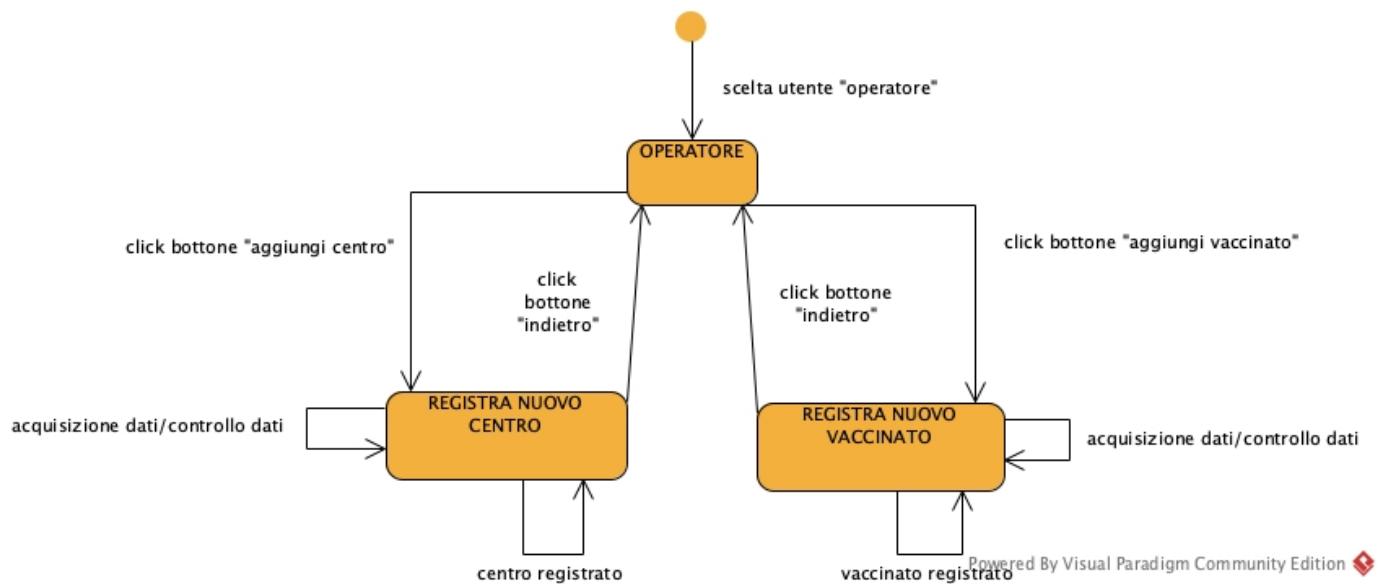
- **STATE DIAGRAM** → si tratta di una presentazione di automi a stati finiti, rappresentanti un sistema, come ad esempio un oggetto di una classe, in termini di:
 - Eventi a cui il sistema è sensibile.
 - Azioni prodotte.
 - Transazioni di stato.

Il funzionamento è molto semplice: un evento istantaneo generato da un'azione da parte di un attore, fa cambiare lo stato dell'oggetto, il quale perdura nel tempo finché un nuovo evento non si verifica.

- Operatore Vaccinale:



- Cittadino:



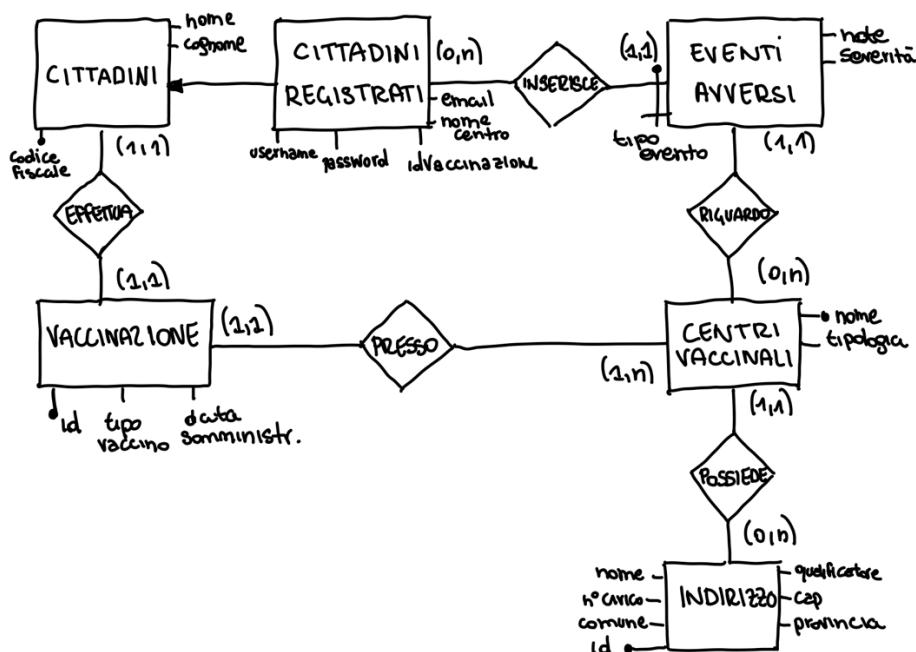
PROGETTAZIONE DEL DATABASE

In questa sezione verranno riportati gli elementi essenziali alla progettazione del database, in particolare secondo i requisiti di sistema è necessario memorizzare informazioni sottoforma di dati riguardanti:

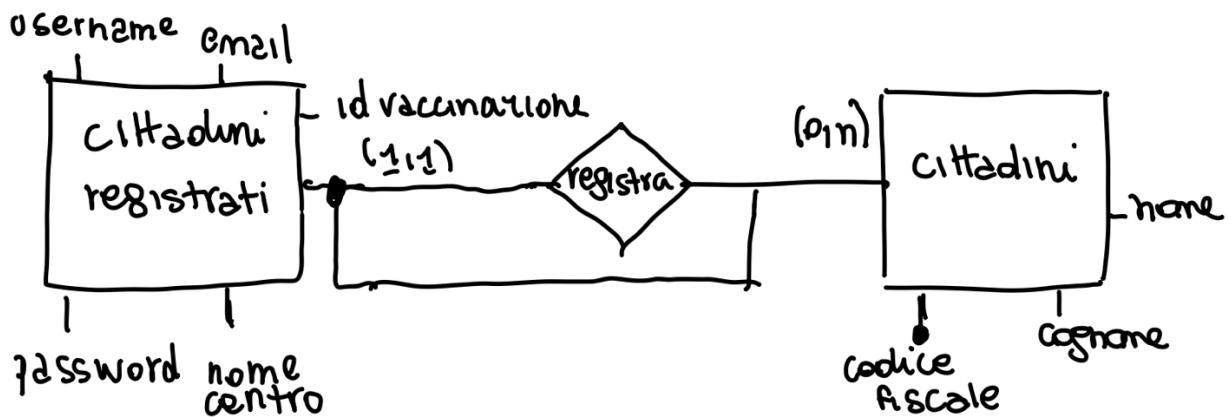
- Identificatori dei centri vaccinali.
- Identificatori dei cittadini vaccinati.
- Registrazione dei cittadini vaccinati.
- Elenco eventi avversi.

Al fine di elaborare nei migliori dei modi la progettazione del database contenente le informazioni essenziali al funzionamento del programma, sono stati eseguiti diversi passaggi, nello specifico:

- **SCHEMA CONCETTUALE** → di seguito verrà introdotto lo schema concettuale dell'applicazione, ovvero un modello di progettazione utilizzato per pianificare e rappresentare visivamente la struttura delle informazioni contenute nel database. Delinea le entità specifiche del sistema, in particolare:
 - Entità
 - Attributi delle entità
 - Relazioni tra entità



- **RISTRUTTURAZIONE DEL RISULTATO** → in questo passaggio viene generato uno schema ER semplificato, ma equivalente a quello di partenza, al fine di semplificare la traduzione successiva.
Nello schema ER precedente è presente una gerarchia di generalizzazione che coinvolge Cittadini e Cittadini Registrati, siccome questa non è direttamente rappresentabile nel modello relazionale, viene sostituita nel modo seguente, generando un nuovo risultato:



- **SCHEMA LOGICO** → Scelto uno specifico DBMS, nel nostro caso PostgreSQL con tool grafico "PgAdmin", è necessario trasmettere i vincoli logici che si applicano ai dati archiviati, traducendo lo schema concettuale sopra rappresentato in uno schema logico per il DBMS prescelto.

- Cittadini (codice fiscale, cognome, nome)
- Indirizzo (id, qualificatore, nome, numeroCivico, comune, cap, provincia)
- Centri Vaccinali (nome, tipologia, indirizzo^{indirizzo})
- Vaccinazione (id, codice fiscale^{cittadini}, data, tipoVaccino, indosi, nomeCentro^{centri Vaccinali})
- Cittadini Registrati (codice fiscale^{cittadini}, username, password, email, id Vaccinazione^{vaccino})
- Eventi Avvorsi (codice fiscale^{cittadini registrati}, evento, severità, note, nomeCentro^{centri Vaccina})

- **VINCOLI DI DOMINIO E INTEGRITA'** → rappresentano vincoli che esprimono condizioni di correttezza nei dati delle tabelle, qualsiasi applicazione che accede deve riconoscere questi vincoli, in base alle informazioni del dominio. Nel sistema progettato, i vincoli sono i seguenti:
 - Nella relazione **CentriVaccinali**:
 - L'attributo nome rappresenta la chiave primaria.
 - L'attributo tipologia ha come dominio i seguenti valori: {aziendale, hub, ospedaliero}.
 - L'attributo idIndirizzo riferisce all'attributo id della relazione **Indirizzo**.
 - Nella relazione **Indirizzo**:
 - L'attributo id rappresenta la chiave primaria.
 - L'attributo qualificatore ha come dominio i seguenti valori: {via, viale, piazza}.
 - L'attributo cap ha come dominio il range di valori compreso: [0, 99999].
 - Nella relazione **Cittadini**:
 - L'attributo codiceFiscale rappresenta la chiave primaria.
 - Nella relazione **CittadiniRegistrati**:
 - L'attributo codiceFiscale è la chiave primaria e riferisce all'attributo codiceFiscale della relazione **Cittadini**.
 - L'attributo idVaccinazione riferisce all'attributo id della relazione **Vaccinazione**.
 - Nella relazione **Vaccinazione**:
 - Gli attributi id e codiceFiscale rappresentano la chiave primaria.
 - L'attributo id è numerico definito su 16 bit.
 - L'attributo codiceFiscale riferisce all'attributo codiceFiscale della relazione **Cittadini**.
 - L'attributo tipoVaccino ha come dominio i seguenti valori: {Pfizer, Moderna, J&J, AstraZeneca}.
 - L'attributo nDosi ha come dominio i seguenti valori: {Prima, Seconda, Terza o Successiva}.
 - L'attributo nomeCentro riferisce all'attributo nome della relazione **CentriVaccinali**.

- Nella relazione **EventiAvversi**:
 - Gli attributi codiceFiscale e evento formano la chiave primaria.
 - L'attributo codiceFiscale riferisce all'attributo codiceFiscale della relazione **CittadiniRegistrati**.
 - L'attributo severità ha come dominio il range di valori compreso: [1,5].
 - L'attributo note può contenere massimo da 0 a 255 caratteri.
 - L'attributo nomeCentro riferisce all'attributo nomeCentro della relazione **CentriVaccinali**.

STRUTTURA GENERALE DELLE CLASSI

Il progetto è strutturato in due suddivisioni: “**ClientLab**”, che si occupa della gestione grafica e di elaborazione/visualizzazione dei dati lato client e “**ServerLab**” che svolge il medesimo compito lato server.

- **CLIENT LAB:**

- Elaborazione e visualizzazione Dati:
 - ClientMain (main)
 - View <Interfaccia>
 - ViewImpl
 - Model <Interfaccia>
 - ModelImpl
 - ControllerImpl
 - Proxy
 - ServerInterface <Interfaccia>
- Gestione Grafica:
 - SceltaOperatoreGUI
 - GUIScambioIP
 - OperatoreGUI
 - RegistraCentroGUI
 - RegistraVaccinatoGUI
 - CittadinoGUI
 - RegisterGUI
 - CercaCentroGUI
 - ElencoCentriGUI
 - LoginGUI
 - EventiAvversiGUI
 - VisualizzaEventiAvversiGUI
 - Colori

- **SERVER LAB:**

- Elaborazione e visualizzazione dati:
 - ServerMain (main)
 - ServerWorker
 - ViewInterface <interfaccia>
 - ModelInterface <interfaccia>
 - ModelImpl
 - Controller
 - SkeletonInterface <interfaccia>
 - DataBaseConnessione
 - EsecutoreQuery
- Gestione Grafica:
 - ServerGUI

CLIENT LAB

Di seguito verranno elencate le classi lato client, con una breve descrizione di carattere architetturale e implementativo.

CLIENTMAIN

La classe principale che contiene il metodo **public static void main (String[] args) {}**, necessario per generare i componenti utili al funzionamento del modulo ClientLab.

VIEW

Interfaccia che fornisce l'overload dei metodi alle classi che la implementano.

VIEWIMPL

Questa classe estende JFrame ed implementa l'interfaccia View. Si occupa di gestire tutte le componenti grafiche dell'applicazione, aggiornando anche i singoli elementi, in base ai dati ricevuti nel metodo **updateView ()**.

MODEL

Interfaccia che fornisce l'overload dei metodi alle classi che la implementano.

MODELIMPL

Questa classe implementa l'interfaccia Model.

Il costruttore salva un riferimento di tipo View passato come parametro e istanzia un riferimento di tipo ServerInterface.

È dotata di un metodo denominato **updateModel ()** che riceve come parametri in input:

- Source → Il nome dell'oggetto che ha generato l'evento.
- Dati → inseriti dall'utente tramite interfaccia.

Dopo aver verificato l'oggetto che generato l'evento chiama i propri metodi secondo le condizioni elencate.

Con i dati ricevuti dalla classe Controller, dopo averli elaborati rendendoli compatibili con il formato richiesto, richiama i metodi di ServerInterface che si occuperanno di completare le operazioni richieste.

Infine, grazie al riferimento di tipo View, viene richiamato il metodo **updateView ()** utile per aggiornare, in base alle informazioni passate come parametro, le componenti grafiche.

CONTROLLERIMPL

Implementa l'interfaccia ActionListener.

Il costruttore salva i riferimenti di Model e View passati come parametri, e si occupa di catturare e aggiungere ai componenti grafici la possibilità di generare eventi.

All'interno di essa è presente il metodo **actionPerformed ()** che prende in input:

- **e** → di tipo ActionEvent, ossia l'evento generato.

Successivamente estrapola il nome dell'oggetto verificando l'istanza di appartenenza e seguendo un serie di condizioni, ricava le informazioni inserite nella View.

Fatto ciò, richiama il metodo del model, che come descritto in precedenza elaborerà i dati sul bottone d'origine.

SERVERINTERFACE

Interfaccia contenente la definizione delle operazioni offerte dal server.

PROXY

Implementa l'interfaccia ServerInterface.

La comunicazione tra client e server è gestita da questa classe, la quale fa da intermediario tra le esigenze del client e la ricerca di risorse sul server, fornendo dei metodi che per ogni specifica funzionalità, inviano, sottoforma di stringhe le informazioni richieste, le quali viaggiano su dei buffer di tipo

ObjectInput e **ObjectOutput**, in particolare:

- **writeObject ()** → metodo che scrive dati sul buffer per essere letti dal server.
- **readObject ()** → metodo che legge i dati inviati dal server sul buffer.

CLASSI GRAFICHE

Classi che gestiscono solo i componenti grafici delle singole schermate.

Dotate di oggetti di tipo JFrame, JPanel, JLabel, JButton ecc.

SERVER LAB

Di seguito verranno elencate le classi lato server, con una breve descrizione di carattere architetturale e implementativo.

SERVERMAIN

La classe principale che contiene il metodo **public static void main (String[] args) {}**, necessario per generare i componenti utili al funzionamento del modulo ServerLab.

VIEWINTERFACE

Interfaccia che fornisce l'overload dei metodi alle classi che la implementano.

SERVERGUI

Implementa l'interfaccia ViewInterface.

Questa classe si occupa di gestire i componenti grafici della schermata di avvio del server.

Contiene il metodo **updateView ()**, necessario ad aggiornare lo stato della schermata in base al valore del campo dati passato come parametro.

MODELINTERFACE

Interfaccia che fornisce l'overload dei metodi alle classi che la implementano.

MODELIMPL

Implementa l'interfaccia model.

Il costruttore salva un riferimento di tipo View passato come parametro e istanzia un riferimento di tipo SkeletonInterface, che si occuperà della comunicazione con il Server, gestendo le richieste.

La classe è dotata del metodo **avvioServer ()**, il quale attende la connessione del client e lancia un thread di gestione dei task ogni qual volta qualcuno si connette.

È presente, inoltre, il metodo **updateModel ()**, che prende in input i dati elaborati dal controller.

Questi, dopo essere stati modellati secondo le compatibilità richieste (generalmente salvati su liste), vengono passate come parametri ai metodi di ServerInterface, in grado di completare le operazioni.

Infine, viene richiamato il metodo **updateView ()** con lo scopo di aggiornare le componenti grafiche.

CONTROLLER

Questa classe implementa l'interfaccia `actionListener`.

Il costruttore salva i riferimenti di Model e View passati come parametri, e si occupa di catturare e aggiungere ai componenti grafici la possibilità di generare eventi.

All'interno di essa è presente il metodo **`actionPerformed()`**, ereditato da `ActionListener` che prende in input:

- `e` → di tipo `ActionEvent`, ossia l'evento generato.

Successivamente estrapola il nome dell'oggetto (in questo caso si occupa di intercettare solo il bottone di avvio del server), ricavando le informazioni inserite nella schermata principale `ServerGUI`.

Fatto ciò, richiama il metodo del model, che come descritto in precedenza elaborerà i dati sul bottone d'origine.

SKELETONINTERFACE

Questa interfaccia contiene la definizione dei metodi necessari al funzionamento del sistema.

SERVERWORKER

Estende la classe `Thread`.

Al costruttore vengono passati come parametri:

- `socket` → connessione con il client.
- `swi` → riferimento di tipo `ServerInterface`, condiviso con le altre istanze di `ServerWorker`.

I thread vengono lanciati, con il metodo **`start()`**, ereditato dal superclasse, quando un nuovo client si connette, in modo tale da gestire in parallelo operazioni da parte di più client.

Viene inoltre ereditato il metodo **`run()`** il quale, si occupa attraverso il metodo **`readObject()`** richiamato su un oggetto di tipo `ObjectInput` di leggere i dati inviati dal client, elaborare le richieste sulla risorsa condivisa di tipo `ServerInterface`, ed infine, inviare l'esito attraverso il metodo **`writeObject()`**, sempre di proprietà di `ObjectOutput`.

DATABASECONNECTION

Classe che fornisce riferimenti di tipo:

- `connection`
- `statement`

Attraverso il metodo **`getConnection()`** si occupa della connessione al database di PostgreSQL dove sarà gestita la parte back-end dell'applicazione.

La connessione avverrà mediante username e password del DB, specificando la porta dove questo girerà.

Il metodo **getStatement ()** svolge la funzione di restituire l'oggetto statement appartenente all'oggetto connection.

Lo statement eseguirà l'interrogazione al DB sottoforma di query.

ESECUTOREQUERY

Questa classe implementa l'interfaccia SkeletonInterface.

Il costruttore salva riferimenti di tipo Connection e Statement relativi alla connessione con il Database.

ModellImpl crea un'istanza di questa classe e fornisce il riferimento dell'oggetto appena creato a tutte le istanze della classe ServerWorker. Il compito principale è quello di fornire tutti i metodi necessari al funzionamento del sistema, ogni metodo, in base alle esigenze da parte del client, richiede come parametri i dati necessari all'esecuzione delle query, che completerà grazie al metodo **executeQuery ()** dell'oggetto di tipo Statement, ritornando un valore in base allo stato finale. (successo o insuccesso).

STRUTTURE DATI UTILIZZATE

Per salvare ed elaborare i dati necessari al funzionamento dell'applicazione, in particolar modo per la comunicazione tra client e server, vengono utilizzate strutture dati differenti, di seguito verranno elencate le principali:

- **ClientLAB – Gestione richieste** → viene inviato in input un comando di “riconoscimento” dell’operazione da svolgere al server, seguita dall’invio delle informazioni, se necessarie, per elaborare la richiesta finale.
Questo avviene mediante la scrittura di valori di tipo String su un buffer dedicato alla comunicazione.
- **ClientLAB – Strutture dati** → il client non ha particolari necessità di memorizzare dati particolarmente strutturati, in quanto questi vengono gestiti dal database, aggiornandoli e mostrandoli in base alla schermata visualizzata. Viene fatta eccezione nel caso il client acceda al form di visualizzazione dei propri eventi avversi: qui i dati vengono salvati su una JTable, componente grafica che riceve come parametri due array:
 - Array monodimensionale → contiene i nomi delle colonne.
 - Array bidimensionale → contiene i valori da inserire nella tabella.

Il popolamento della tabella avviene nel seguente modo:

Effettuato con successo l’inserimento dell’evento avverso, il Model invia una richiesta al server per ottenere l’elenco degli eventi avversi dell’utente

connesso, una volta ottenuti questi vengono inviati alla View, la quale classe salverà l'elenco in un riferimento di tipo `List<List<String>>`.

Siccome i dati non sono compatibili con il formato accettato dalla `JTable`, prima di inserirli, questi valori vengono convertiti, mediante apposito metodo, in un array bidimensionale di stringhe.

- **ServerLAB – gestione richieste** → analogamente al funzionamento per la gestione delle richieste nel modulo Client, il server legge sul buffer dedicato le informazioni inviate, ovvero:
 - `InputClient` → richiesta dell'operazione da eseguire.
 - Singoli Valori → rappresentano le informazioni fondamentali per eseguire i metodi utili al funzionamento del sistema.

Ricevuti questi dati, il server richiamerà il metodo corrispondente dell'istanza della classe `EsecutoreQuery`, inviando infine l'esito elaborato.

NOTA BENE: Per gestire più operazioni in parallelo, le richieste vengono inviate ad ogni `ServerWorker`, ovvero il thread server che gestisce connessioni di client diverse.

- **ServerLAB – Strutture dati** → date le molteplicità di funzioni offerte, il server invia, attraverso le strutture dati necessarie alla memorizzazione, i dati ottenuti dai metodi della classe `EsecutoreQuery`. Di seguito verranno illustrati i possibili tipi di valori di ritorno e strutture:
 - Boolean → valore di verità: ritornato dai metodi che verificano delle condizioni necessarie al funzionamento del sistema, caratterizzati da due esiti: True nel caso di successo, o false di insuccesso.
 - Int → valore intero: ritornato dai metodi che possono elaborare più esiti diversi, viene utilizzato, ad esempio, nel momento in cui l'utente vuole aggiungere un centro vaccinale, un nuovo vaccinato, un evento avverso ecc.
 - Un valore positivo rappresenta l'operazione andata a buon fine.
 - Un valore nullo indica che l'operazione non è stata ultimata ma non ha generato errori.
 - Un valore negativo comporta la generazione di un errore e il conseguente insuccesso dell'operazione.
 - String → valore ritornato dai metodi che lavorano su operazioni di richiesta di dati specifici.
 - List <String> → lista di Stringhe: ritornata nel momento in cui l'utente richiede l'accesso a più informazioni relative al metodo specifico.
 - List <List<String>> → questa struttura dati corrisponde ad una Lista contenente un'altra lista di stringhe. Questo è utile nel caso si

voglia memorizzare più elementi non specifici, dove ogni elemento contiene le informazioni relative al tipo di operazione da effettuare. Un esempio sono i dati corrispondenti agli eventi avversi di un utente connesso, dove nella lista più esterna è contenuto il riferimento all'utente connesso, il quale punta all'indice della lista più interna, contenente gli effettivi dati da visualizzare.

SCELTE ALGORITMICHE

Il sistema è stato sviluppando seguendo determinate scelte algoritmiche, implementate in base alle caratteristiche dell'applicazione, in particolare verrà illustrato:

- Comunicazione Client-Server.
 - Invio richieste lato client.
 - Gestione dei task lato server.
 - Gestione risorsa condivisa.
 - Elaborazione ed invio dati ai client.
-
- **Comunicazione Client-Server** → l'interazione, basata su protocollo TCP sfrutta "Socket": oggetto software "connection-oriented" che permette l'invio e la ricezione dei dati tra host remoti o tra processi locali. Più precisamente, il concetto di socket si basa sul modello Input/Output, quindi sulle operazioni di open, read, write e close. L'utilizzo, infatti, avviene secondo le stesse modalità, aggiungendo i parametri utili alla comunicazione, quali:
 - Indirizzi
 - Numeri di porta
 - Protocolli

NOTA BENE: Socket in comunicazione formano una coppia composta da Indirizzo e porta di client e server, instaurando tra di loro una connessione logica.

Di seguito verrà illustrato nel dettaglio il compito di ciascuna classe interessata:

ServerLAB – ModelImpl: classe progettata per avviare il server attraverso l'apposito metodo **avvioServer()**, ricevendo come parametro le informazioni necessarie alla connessione al Database.

```
public void avvioServer(Object dati) throws SQLException {
    try {
        theServerSocket = new ServerSocket(ModelImpl.PORT); // porta di ascolto
        System.out.println("Server Started... waiting connection");
        swi.creazioneTabelle();
        boolean loop = true;

        while (loop) {
            socket = theServerSocket.accept(); // attende connessione --> ritorna oggetto socket
            System.out.println("Server: connessione accettata " + socket);
            new ServerWorker(socket, swi); // lancia un server thread che gestisce task
        }
    } catch (IOException e) {
        e.printStackTrace();
        System.err.println("MODEL: Server non avviato: " + e.toString());
    }
}
```

- Socket theServerSocket → il server crea il rispettivo socket, ponendolo in ascolto su una determinata porta.

NOTA BENE: possiamo avere numeri di porta diversi tra client e server, perché una potrebbe essere dedicata solo al traffico in uscita, l'altra solo in entrata, dipende dalla configurazione dell'host.

- **theServerSocket.accept()** → metodo che attende la richiesta da parte di un client, nel caso in cui sia accettata, viene creata una nuova connessione ritornando il socket client connesso. Ora client e server comunicano attraverso un canale virtuale, creato appositamente per il flusso dei dati.
- ServerWorker swi → classe thread che si occupa di gestire richieste da parte dei client, creata ogni qualvolta questi si connettono. Come argomenti il costruttore riceve:
 - Socket : univoco per ogni client
 - Swi → risorsa condivisa, necessaria alla gestione delle richieste ed al corretto funzionamento del sistema.

NOTA BENE: queste istruzioni in codice sono “racchiuse” in un ciclo **while(true)**, questo per permettere la connessione e la gestione delle differenti richieste in parallelo, svolte da più entità.

ClientLAB – Proxy: funzionamento analogo lato client, in particolare:

```
this.socket = new Socket(addr, port); //indirizzo e porta server  
this.out = new ObjectOutputStream(this.socket.getOutputStream());  
this.in = new ObjectInputStream(this.socket.getInputStream());
```

- Socket socket → Il client crea il rispettivo socket, a differenza del server, accetta due parametri, passati come argomenti nel costruttore della classe:
 - o Addr: indirizzo IP relativo all'host connesso (di default "localhost").
 - o Port: porta di ascolto.
- ObjectInputStream in → creazione stream dove saranno letti i flussi di dati in entrata.
- ObjectOutputStream out → creazione stream dove saranno scritti i flussi di dati in uscita.

NOTA BENE: il sistema è stato progettato per leggere e scrivere Stringhe di valori, per quanto riguarda le informazioni relative al funzionamento dell'applicazione, e di valori interi e booleani per esiti di verifica.

Proxy funge da intermediario tra richieste del client e ricerca delle risorse sul client, pertanto, l'istanza proxy verrà costruita nella classe ModelImpl in modo tale da eseguire chiamate locali su questo oggetto.

```
this.proxy = new Proxy(InetAddress.getByName("localhost"), 8080); //local host e porta
```

Dove:

- "localhost" → rappresenta indirizzo (addr).
 - 8080 → porta di ascolto.
- **Invio richieste lato Client** → stabilità la coppia connessa e inizializzati gli stream di flusso dei dati, il client ed il server sono pronti per iniziare la vera e propria comunicazione.

Di seguito verranno analizzati i compiti svolti da ogni singola classe interessata alla gestione dello scambio di informazioni:

ClientLAB – Proxy: si occupa di gestire le richieste del client ed inviarle al server, in base all'informazione/operazione che l'host necessita di ottenere/eseguire.

```
public int registraCentroVaccinale(String nome, String qualificatore, String indirizzo, String
Integer res = 0; //segnaile restituito dal server con callBack

try {
    this.out.writeObject("registraCentro");
    this.out.writeObject(nome);
    this.out.writeObject(qualificatore);
    this.out.writeObject(indirizzo);
    this.out.writeObject(numeroCivico);
    this.out.writeObject(comune);
    this.out.writeObject(provincia);
    this.out.writeObject(cap);
    this.out.writeObject(tipologia);

    res = (Integer) this.in.readObject(); //ascolta risposta da server

} catch (IOException e) {
    System.err.println("Proxy: problemi nell'acquisizione: " + e.toString());
} catch (ClassNotFoundException e) {
    System.err.println("Proxy: problemi con salvataggio: " + e.toString());
}

return res.intValue(); //se maggiore di zero aggiunto, se = 0 < no perchè già esiste
}
```

Considerando la molteplicità di servizi a disposizione, verrà analizzato un esempio per descrivere il funzionamento. Il metodo in questione si occupa di inviare la richiesta di registrare un nuovo centro vaccinale; nel dettaglio:

- **writeObject ()** → eseguito sullo stream in uscita (out): questo metodo è responsabile della scrittura del/i dato/i da inviare al server.
- **readObject ()** → eseguito sullo stream in entrata (in): questo metodo si occupa di leggere la risposta inviata dal server, ottenendo l'informazione necessaria al funzionamento del sistema.

NOTA BENE: ai fini del corretto funzionamento, per ogni dato inviato, la controparte prima di svolgere qualsiasi funzione, dovrà implementare l'apposito metodo di lettura, evitando di sollevare eccezioni o generare errori.

- **Gestione task lato Server** → il modulo server riceve le richieste da parte dei client connessi, generando un thread che si occupa di interpretare i comandi e raccogliere le informazioni.

In particolare, questa funzionalità è implementata dalla classe:

ServerLAB – ServerWorker → all'avvio del thread con il metodo start (), viene eseguito il run (), ereditato dalla superclasse thread, rimanendo così in ascolto delle richieste lato client ed eseguendo la parte di codice interessata ai fini di fornire la corretta soluzione al servizio ricercato.

```

public void run() {
    String inputClient;

    while(true) {
        try {
            inputClient = (String) this.in.readObject();

            if(inputClient.equals("registraCentro")) {
                String nome = (String) this.in.readObject();
                String qualificatore = (String) this.in.readObject();
                String indirizzo = (String) this.in.readObject();
                String numeroCivico = (String) this.in.readObject();
                String comune = (String) this.in.readObject();
                String provincia = (String) this.in.readObject();
                String cap = (String) this.in.readObject();
                String tipologia = (String) this.in.readObject();

                //chiamata il metodo per eseguire query
                int esito = this.swi.registraCentroVaccinale(nome, qualificatore, indirizzo,
                this.out.writeObject(esito); //comunica al client esito
            }
        }
    }
}

```

Considerando la molteplicità di servizi a disposizione, verrà analizzato un esempio per descrivere il funzionamento. Il metodo in questione si occupa di registrare un nuovo centro vaccinale; nel dettaglio:

- InputClient → stringa che rappresenta quale esecuzione richiesta dal client deve essere eseguita dal server.
- ServerInterface swi → oggetto di tipo ServerInterface, necessaria per accedere alla risorsa condivisa per registrare concretamente sul Database il nuovo centro vaccinale, grazie ai metodi messi a disposizione dall'interfaccia server.
- La ricezione e l'invio dell'esito avviene nel medesimo modo sia lato client che lato server. In base alle informazioni da estrapolare ed inviare vengono utilizzati, come riportato in precedenza, i metodi **readObject ()** e **writeObject ()** appartenenti rispettivamente alla classe InputObject e OutputObject.
- **Gestione risorsa condivisa** → recuperati i dati necessari al corretto funzionamento del sistema, il server deve occuparsi, attraverso appositi strumenti, di effettuare concretamente le operazioni richieste lato client.
Considerando il precedente esempio, è bene notare che il server thread richiama un metodo appartenente a server interface, in particolare:

```
//chiamata il metodo per eseguire query
int esito = this.swi.registraCentroVaccinale(nome, qualificatore, indirizzo, numeroCivico, comune, provincia, cap, tipologia);
```

- Il metodo richiede come argomenti i corretti dati al fine di richiamare il metodo con successo.

La classe che si occupa di effettuare le operazioni di salvataggio, manipolazione e controllo delle informazioni è la seguente:

ServerLAB – EsecutoreQuery → implementa l'interfaccia Skeleton Interface, che fornisce i metodi necessari all'esecuzione dei processi.

Questa classe esegue localmente i metodi interfacciandosi con il Database. Terminata l'esecuzione, il server memorizza il risultato in una variabile intera "esito", che ha il compito principale di inviare la risposta al client, assumendo comportamenti diversi in base al suo valore.

Di seguito verrà illustrato un esempio dove il metodo in questione si occupa di verificare, attraverso un'interrogazione al database mediante query, l'esistenza di un centro vaccinale ricevendo come parametro il nome che lo identifica.

```
public synchronized boolean esisteCentroNome(String nomeCentro) throws SQLException {  
    boolean ret = false;  
    String query = "SELECT nome FROM centriavaccinali WHERE nome = '"+ nomeCentro +"';";  
    ResultSet rs = istruzione.executeQuery(query);  
  
    while(rs.next()) {  
        ret = true;  
    }  
  
    return ret;  
}
```

NOTA BENE: nella segnatura del metodo è presente la keyword **synchronized**, in quanto l'istanza di EsecutoreQuery sarà un oggetto condiviso tra tutte le istanze di ServerWorker. Risulta quindi fondamentale in modo da prevenire eventuali **race condition** che possono verificarsi.

- **Elaborazione ed invio dati ai client** → le informazioni lavorate dal server devono successivamente essere rese visibili sull'interfaccia client, in modo che entrambi riescano a compiere le operazioni nel modo corretto, secondo le regole stabilite. Questo avviene mediante dei meccanismi appositamente progettati ed implementati, che coinvolgono classi diverse, in particolare:

ClientLab – ModelImpl: questa classe implementa i metodi necessari al funzionamento della logica applicativa, interfacciandosi localmente con il proxy, in modo che quest'ultimo poi invia e riceve, come riportato precedentemente, i dati necessari, in base alla funzionalità scelta. Quando il server invia i dati, il model si occupa localmente, attraverso apposite chiamate, di estrarre i dati ed indirizzarli nel modo corretto. In particolare, il metodo **updateModel ()**, viene eseguito ogni volta generato un evento, ed in base all'evento identificato si occupa di:

- Recuperare i dati dal database
- Organizzarli per il model
-

Di seguito verrà illustrato un esempio, dove l'evento generato dall'utente corrisponde alla funzione di registrazione di un nuovo centro vaccinale:

```
public void updateModel(String source, Object dati) {
    String button = source;
    Object datiPerModel = null;
    this.flag = false;

    if(button.equals("REGISTRA CENTRO")) {
        List<String> datiCentro = (List<String>) dati;
        datiPerModel = (List<String>) registraCentroVaccinale(datiCentro);

        if(((List<String>) datiPerModel).get(0).equals("ERRORE:")){
            this.flag = true;
        }
    }
}
```

- Dati → informazioni ricevute dalla view di registrazione del centro.
- DatiPerModel → inviati dal server e modellati secondo il formato compatibile.

Dopo aver eseguito le funzioni interne alla classe, per elaborazione e controllo dei dati, questi ultimi sono pronti per essere visualizzati dall'utente in base al servizio richiesto.

Viene così chiamato il metodo **updateView ()** che si occuperà di inviare i dati alla schermata, che, come nel caso del model, eseguirà successivamente le formattazioni necessarie per la visualizzazione.

```
this.v.updateView(button, datiPerModel, this.flag); //aggiorna view in base all'elaborazione
```

CONNESSIONE A DATABASE

L'applicazione sviluppata integra una connessione al database, in particolare a PostgreSQL, dove verranno salvati i dati necessari al funzionamento del sistema e dove mediante apposite interrogazioni il server estrapolerà i dati richiesti.

In particolare, il funzionamento del meccanismo di connessione viene eseguito seguendo determinati passaggi e coinvolgendo classi differenti, illustrate di seguito:

ServerLAB – ServerGUI → interfaccia grafica che si occupa di gestire i principali campi di connessione al DB prima dell'avvio del server, in particolare vengono specificati:

- Username
- Password
- Host
- Porta
- NomeDB

NOTA BENE: i valori di questi campi sono generati al primo accesso in pgAdmin, tool grafico di gestione del DB.

NOTA BENE: valori di host e porta sono differenti dai valori specificati per client e server, perché i primi si occupano di istaurare una connessione con il DB, e gli altri per comunicare scambiando informazioni, quindi svolgono operazioni differenti.

ServerLAB – DataBaseConnessione → questa classe si occupa della vera e propria connessione al database, implementa il pattern Singleton, che, come da definizione, richiederà di istanziare il costruttore vuoto e i metodi effettivi di connessione, ovvero:

- **getConnection ()** → se la connessione non è già stabilita, restituisce un oggetto di tipo Connection, il quale si riferisce alla connessione appena instanziata. È importante sottolineare quali parametri riceve in ingresso:
 - user: username con cui accedere al DB.
 - Pass: password con cui accedere al DB
 - Host2: indirizzo della macchina su cui è presente il DB.
 - Port: porta dove gira il DB.
 - Name: nome del DB.

- **getConnectionServer ()** → se la connessione non è già stabilita, ritorna una oggetto di tipo Connection. Questo metodo è utilizzato per effettuare autenticazione ai server di PostgreSQL, per poi eseguire effettivamente la creazione, qual'ora non sia già presente, del database e la concreta connessione. Riceve come argomenti gli stessi del precedente metodo, escluso appunto il nome del DataBase.
- **getStatement ()** → metodo che restituisce un oggetto di tipo statement all'oggetto connection instanziato, attraverso il proprio metodo **createStatement ()**. Necessario per l'esecuzione delle query.

La classe implementa inoltre i metodi **closeStatement ()** e **closeConnection ()** per chiudere la connessione verso il DB istanziato.

ServerLAB – ModelImpl → al click del bottone di avvio del server, la view richiama il metodo **updateModel ()**, che si occupa di salvare i riferimenti dei campi sopra specificati in una lista, passandoli come parametro ad una nuova istanza di EsecutoreQuery, utili per istaurare connessione per la manipolazione dei dati.

```
public void updateModel(Object dati) throws SQLException {
    List<String> datiPrelevatidaArgomento = (List) dati; // preleva dati passati dal controller
    this.v.updateView("avvio"); // aggiorna view in base all'elaborazione
    if (datiPrelevatidaArgomento.get(0).equals("AVVIO SERVER")) {
        this.v.updateView("avvio"); // aggiorna view in base all'elaborazione
        List<String> lista = (List) dati;
        swi = new EsecutoreQuery(lista.get(1), lista.get(2), lista.get(3), lista.get(4),
                               lista.get(5).toLowerCase());
        avvioServer(null); // avvia il server
    }
}
```

Dopo il salvataggio dei dati, verrà richiamata la funzione avvioServer() della medesima classe.

ServerLAB – EsecutoreQuery → I metodi elencati, presenti nella classe DatabaseConnessione, verranno richiamati nel costruttore della classe EsecutoreQuery, istanziata precedentemente:

- Connessione → richiama il metodo di connessione al DB.
- Istruzione → verrà creato l'oggetto Statement, necessario per eseguire le query all'interno del DB.

```

public EsecutoreQuery(String username, String password, String host, String port, String nomeDB) {
    try {
        this.nomeDataB = nomeDB;
        LoginServer(host, port, username, password);
        creazioneDB();
        this.conessione = DataBaseConnessione.getConnection(username, password, host, port, nomeDB); //prende connessione al database
        this.istruzione = (Statement) conessione.createStatement(); //statement per eseguire query
    } catch (SQLException e) {
        JOptionPane.showMessageDialog(null, "CONNESSIONE FALLITA", "SERVER", JOptionPane.ERROR_MESSAGE);
        System.out.println("ESECUTORE QUERY: connessione al DB non riuscita " + e.toString());
    }
}

```

NOTA BENE: Prima della creazione del DB e l'effettiva connessione, si procede ad effettuare l'autenticazione e connessione al server di PostgreSQL, con l'apposito metodo **LoginServer ()**, che accetta come argomenti host, porta, username e password e istanzia un oggetto di tipo Connection richiamando metodo **getConnectionServer ()** della classe DataBaseConnessione.

```

public void LoginServer(String host, String port, String username, String password){
    try {
        this.conn = DataBaseConnessione.getConnectionServer(username, password, host, port); //prende connessione al server
        this.istr = (Statement) conn.createStatement();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

Completati questi passaggi, ora il nostro sistema è pronto ad eseguire tutte le operazioni nel modo corretto, interfacciandosi con tutti gli elementi necessari. Contrariamente, nel caso in cui durante la connessione si riscontra qualche problema, l'errore verrà catturato con un'eccezione e mostrato nel terminale del proprio IDE di sviluppo.

Errori comuni si possono presentare qualora:

- Valore dei campi di autenticazione e connessione errati.
- Login al server PostgreSQL fallito.
- Database già esistente: in questo caso catturata l'eccezione, il sistema continuerà a funzionare, elaborando i dati sul database già esistente.

NOTA BENE: alla prima creazione del DB, verranno create, attraverso comandi in linguaggio SQL, le tabelle necessarie al funzionamento dell'applicazione. Nei successivi accessi, se queste già presenti, non verranno generate.

PATTERN UTILIZZATI

“**Keep it simple stupid**”: questa frase, coniata da Kelly Johnson, delinea in modo intuitivo il significato di pattern.

Nello specifico, rappresentano schemi di soluzioni riutilizzabili quando si incontrano, nello sviluppo di progetti software, problemi grossi, ricorrenti e prevedibili.

Nella sezione seguente verranno illustrati i principali pattern implementati nella struttura dell'applicazione, descrivendo:

- Pattern MVC (Model View Controller) per Client e Server.
 - Pattern singleton per connessione a DB.
 - Proxy per modulo Client.
 - Skeleton per modulo Server.
-
- **PATTERN MVC** → acronimo di **Model View Controller**. Appartiene alla famiglia di pattern di tipo architetturali e viene utilizzato per la gestione dei componenti del sistema, particolarmente indicato per la realizzazione di applicazioni grafiche. Il suo funzionamento comporta il disaccoppiamento di viste e modelli, con il vantaggio che l'aspetto degli oggetti possa essere modificato indipendentemente dalla logica applicativa. Il pattern è implementato mediante tre componenti:
 - **View** → mediante interfaccia View, implementata dalla classe ViewImpl.
 - **Ruolo all'interno del sistema:** gestione dei componenti grafici attraverso creazione, aggiornamento e visualizzazione sia dei componenti stessi che delle singole schermate, fornendo informazioni legate a quella particolare view. Per garantire l'indipendenza della view, è stata definita un'interfaccia View, che contiene la definizione dei metodi necessari al funzionamento dei componenti all'interno dell'applicazione. Così facendo, per ogni nuova view creata, (nel nostro sistema rappresentato dalla parola GUI) risulta necessario implementare soltanto l'interfaccia View per garantire il corretto funzionamento. La view riceve attraverso il proprio metodo **updateView ()**, richiamato dal Model, le informazioni necessarie all'aggiornamento dei singoli componenti.
 - **Model** → mediante interfaccia Model, implementata dalla classe ModelImpl.
 - **Ruolo all'interno del sistema:** gestione della logica del sistema. Riceve i dati necessari all'elaborazione da parte del Controller. Elabora il procedimento modellando i dati ricevuti secondo il formato compatibile richiesto, decide le operazioni da eseguire e quali dati inviare alla view. Viene

definita un'interfaccia Model ed un'implementazione ModelImpl, in modo che le operazioni e le info necessarie siano garantite dall'implementazione dell'interfaccia, in modo indipendente. Ottiene un riferimento alla view nel momento che questa viene istanziata. È dotato di un metodo **updateModel ()**, richiamato dalla View, necessario all'invio delle informazioni e alle successive operazioni.

- **Controller** → implementato dalla classe ControllerImpl.
 - **Ruolo all'interno del sistema:** gestione di possibili eventi generati dalla view ed invio delle informazioni necessarie al Model, in base all'evento generato. Implementa l'interfaccia ActionListener, che può essere considerata interfaccia del controller, e ControllerImpl la sua implementazione. Quando viene instanziato, prende i riferimenti alla View e al Model e aggiunge i componenti della view come actionListener con il metodo **addActionListener ()**. Attraverso il metodo **actionPerformed ()**, ereditato dall'interfaccia, che riceve come argomento l'evento generato. Tale metodo si occuperà di ricevere i dati dalla View e di inviarli al Model per effettuare le successive rielaborazioni.
- **PATTERN SINGLETON** → appartiene alla famiglia dei pattern creazionali, i quali forniscono meccanismi per la creazione di oggetti. In particolare, il compito del Singleton è garantire che l'oggetto venga istanziato una sola volta. La classe DatabaseConnessione implementa questo pattern, creando una sola istanza di connessione al database, se questa già non esiste.
- **PATTERN PROXY** → pattern di tipo strutturale, che si occupano di gestire separazione tra interfaccia e implementazione. Nel dettaglio, Proxy si occupa di separare la logica di funzionamento del client dalla parte di comunicazione con il server, comportandosi da intermediario. Nel sistema sviluppato l'integrazione del pattern è così definita:
 - **ModelImpl** → gestisce la logica di funzionamento lato client, ovvero l'elaborazione dei risultati. Per fare ciò, la classe ha bisogno di effettuare delle operazioni sul Database, necessitando quindi di una comunicazione con il Server. Viene così istanziato un riferimento alla classe Proxy, che implementa l'interfaccia ServerInterface, contenente i metodi per ogni operazione che il

server deve eseguire, nascondendo così la comunicazione al client, che si interfacerà con chiamate locali al proxy.

- **Proxy** → Come anticipato, la classe Proxy gestisce le comunicazioni locali con i client, e remote con il server, instaurando una connessione basata su socket. In tal modo, per ogni metodo che il client sceglie di eseguire, il proxy invia la richiesta al server con i parametri necessari ed attende un riscontro.
- **PATTERN SKELETON** → svolge la stessa funzione del Proxy lato Server, in particolare il compito di Skeleton è quello di separare la logica di funzionamento del Server dalla parte di comunicazione con il client. Nel sistema sviluppato, l'integrazione del pattern è così strutturata:
 - **ModelImpl** → si occupa della logica di connessione lato server all'avvio del server, viene istanziata la connessione basata su socket, dove il server rimarrà in ascolto di nuovi client che si vorranno connettere. Per ogni client connesso verrà creato un nuovo Thread.
 - **ServerWorker** → estende la superclasse Thread, gestisce la logica di comunicazione con il client. Il suo compito è, in base alla richiesta del client, di richiamare i metodi corrispondenti dell'istanza EsecutoreQuery, che implementa il funzionamento effettivo del Server. Elaborati i dati, il thread avrà il compito di inviare l'esito al mittente.
 - **SkeletonInterface** → interfaccia che gestisce la logica di funzionamento del server, definendone i prototipi necessari.
 - **EsecutoreQuery** → classe che gestisce la logica di funzionamento del server, implementando i metodi definiti dall'interfaccia Skeleton. Considerando che la logica di funzionamento del server consiste nell'effettuare operazioni con il database, in fase di creazione ottiene il riferimento alla connessione con quest'ultimo.

I principali vantaggi dei Pattern Proxy e Skeleton sono espressi in termini di efficienza e sicurezza.

MAVEN

Maven è uno strumento di Build Automation utilizzato prevalentemente nella gestione di progetti Java.

Il vantaggio derivante da questo tool è da subito evidente: se generalmente per sviluppare un software sono necessarie numerose fasi, attraverso “build automation” l’intero processo viene automatizzato, riducendo il carico di lavoro del programmatore e diminuendo le possibilità di errori.

I componenti principali sono:

- File **pom.xml** → ha il compito di definire identità e struttura del progetto, permettendo di inserire configurazioni, dipendenze e plugin.
- **Goal** → insieme di funzioni che possono essere eseguiti sui diversi progetti.
- Cartella **repository** → l’utente è in grado di gestire il sistema delle librerie.
- File **setting.xml** → usato per la configurazione di repository e profili.

Ai fini del funzionamento del progetto verranno evidenziati le componenti presenti nel file **pom.xml**:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>ClientLAB</artifactId>
  <version>1.0-SNAPSHOT</version>
```

- Project → L’elemento è la radice del descrittore. Nella tabella seguente sono elencati tutti i possibili elementi figlio.
- ModelVersion → Dichiara a quale versione del descrittore del progetto questo POM è conforme.
- groupId → Un identificatore universalmente univoco per un progetto.
- artifactID → L’identificatore per questo artefatto che è univoco all’interno del gruppo specificato dall’ID gruppo. Un artefatto è qualcosa che viene prodotto o usato da un progetto.
- version → La versione attuale dell’artefatto prodotto da questo progetto.

All'interno del tag <dependencies>, vengono aggiunte le dipendenze del progetto, in particolare, ai fini del corretto funzionamento, quelle appartenenti al progetto “Vacciniamo” sono:

- **JCalendar** → componente grafico di visualizzazione del modello di Calendario.
- **Postgresql** → file .jar per permettere la connessione al database di PostgreSQL.
- **JavaDoc** → permette la creazione della JavaDoc, parte integrante della documentazione tecnica.
- **Jar** → consente l'esportazione del progetto sviluppato in file .jar eseguibile.

NOTA BENE: Sia il modulo ClientLAB che il modulo ServerLAB contengono file pom.xml contraddistinti in base alle automazioni che si vogliono implementare.

TESTING

Ai fini del corretto funzionamento dell'intero sistema, sono stati eseguiti test differenti su diverse macchine e su differenti sistemi operativi.

Lo sviluppo è stato supportato da molteplici attività di Debug e analisi. Considerando la caratteristica del software di girare in parallelo e soprattutto evitando race condition tra tutti coloro che svolgono operazioni, il progetto è stato realizzato in modo tale che qualsiasi computer possa funzionare sia da client che da server (implementando ovviamente gli opportuni requisiti specificati nel manuale utente).

Queste operazioni sono garantite da test effettuati in presenza utilizzando i computer dei componenti del team di sviluppo.

Per garantire la massima versatilità ed efficienza dell'applicazione viene messa a disposizione dell'utente client l'opzione che permette di generare un dataset per testare il corretto funzionamento di tutti i servizi a disposizione, senza l'obbligo di inserire manualmente i contenuti.

Ciò è possibile eseguirlo all'avvio del modulo Client, cliccando sull'icona delle impostazioni nella schermata di "Scelta utente" e selezionando il bottone: "Dataset".

SITOGRAFIA

- <https://docs.oracle.com/en/java/>
- <https://maven.apache.org/guides/>
- <https://www.postgresql.org/docs/>
- <https://www.pgadmin.org/docs/>
- <https://www.visual-paradigm.com/support/documents/>
- <https://en.wikipedia.org/wiki/Model–view–controller>
- <https://www.codeupset.com/advantages-of-model-view-controller/>
- <https://freelancersdev.com/advantages-of-using-mvc/>
- <https://en.wikipedia.org/wiki/Entity%E2%88%92relationship>