# MANUALE TECNICO

04/09/2021

v1.0

Ciceri Luigi

# Indice generale

# Introduzione

Lo scopo di questo manuale è di descrivere le scelte architetturali, algoritmiche e di strutture dati utilizzate nello sviluppo del progetto, nonché di fornire una descrizione del funzionamento delle classi.

Per quanto riguarda queste ultime, non verranno illustrate nel dettaglio ma analizzate nelle loro parti che risultano più complesse ed importanti al fine di descrivere al meglio il funzionamento ad alto livello dell'applicazione.

Per ogni altro dettaglio si rimanda alla documentazione *JavaDoc*.

# Librerie esterne

Nel progetto sono state utilizzate delle librerie esterne per velocizzare lo sviluppo e ridurre la complessità del progetto.

## SimpleFlatMapper

Questa libreria permette la lettura e scrittura di file in formato .csv in modo semplice e con delle ottime prestazioni.

Tale libreria permette di gestire automaticamente i casi di escaping dei separatori.

Si è infatti rivelata una tra le più veloci sia in lettura sia in scrittura tra tutte le librerie simili.

La libreria è coperta da licenza MIT.

## OpenJFX 16

La libreria OpenJFX permette lo sviluppo di applicazioni Java con interfaccia grafica e cross-platform.
Nel file binario .jar sono infatti incluse le librerie necessarie ad eseguire il programma su tutti i principali sistemi opereativi desktop (Windows, Linux, MacOS), al fine di semplificare la distribuzione e al contempo rispettare la licenza GPL con classpath exception sotto al quale OpenJFX è rilasciata.

## JFoenix

Contiene componenti aggiuntivi OpenJFX che rispettano le specifice del Material Design creato da Google per applicazione mobile e web.

Essendo questa libreria non aggiornata per l'utilizzo con Java SDK 16, è stato necessario aggiungere alcuni argomenti in fase di esecuzione dell'applicazione, a causa di recenti cambiamenti alle JDK.

Il tema base è inoltre stato leggermente modificato per avvicinarsi ancora di più alle specifiche del Material Design.

La libreria è coperta da licenza MIT.

### Guava core

Contiene utilità e strutture dati aggiuntive. Di particolare interesse nell'ambito del progetto è la struttura dati *LinkedListMultiMap*.

Essa implementa una *LinkedHashMap* che consente di mappare più valori alla stessa chiave.

È coperta da licenza Apache-2.0.

### Ikonli

Fornisce icone in modo facile da utilizzare. In particolare è stato utilizzato il tema Material Design 2 per alcuni elementi dell'interfaccia grafica.

È coperta da licenza Apache-2.0.

## Sistemi di automazione

Nel progetto è stato utilizzato lo strumento di automazione Gradle.

Nonostante le sue numerose capacità, il suo utilizzo si è limitato alla gestione delle dipendenze e alla creazione di artefatti.

Per questo ultimo compito in particolare si è deciso di utilizzare il plugin *Shadow* per creare un "fat jar" contenente tutte le dipendenze necessarie al corretto funzionamento dell'applicazione.

Inoltre grazie al task distribution di Gradle è stato possibile creare degli script di avviamento multi-piattaforma in automatico, nonostante la struttura delle cartelle sia stata leggermente modificata per aderire alle specifiche di progetto.

L'utilizzo di questo strumento ha anche reso possibile la creazione di un progetto indipendente rispetto all'IDE utilizzato.

# Architettura

L’applicazione è stata progettata suddividendo il progetto in 4 aree principali:

- Gestione dati

  La gestione dei dati è affidata alle classi nel package *datamanager* il cui compito è di gestire la lettura e scrittura dei file e l’accesso dei dati da parte di tutte le altre classi

- Interfaccia

  Tutto ciò che riguarda l’interfaccia grafica è contenuto nel package *ui*.

  Il compito delle classi nel suddetto package è di gestire l’interfaccia e raccogliere i dati in essa inseriti, oltre a gestire la navigazione tra pagine

- Gestione operazioni

  La gestione delle operazione e azione è affidata alle classi nei packages *centrivaccinali* e *cittadini.*

  Queste classi rappresentano un layer intermedio tra dati e interfaccia

- Dati

  Le classi in questo package rappresentano oggetti ed enumeratori

## Formato dei file

I file sono salvati in formato csv in modo da poter essere utilizzati anche per altri scopi esterni a questa applicazione (machine learning, analisi statistica, feedback per i produttori dei vaccini, scopi medici, ecc.). Tale formato è inoltre molto diffuso e di facile comprensione.

Per evitare problemi di utilizzo dei dati su piattaforme diverse, tutti i file sono codificati in UTF-8.

Ogni file contiene nella prima riga un header che contiene il numero di oggetti presenti nello stesso.

I file e le cartelle mancanti sono creati se necessario in fase di caricamento, tenendo conto che essi non possano essere eliminati durante l’esecuzione dell’applicazione.

## Tipi enumerativi

I tipi enumerativi implementano un pattern che consente di ottenere una rappresentazione in stringa ben formattata e allo stesso tempo di ottenere un enumeratore a partire da una rappresentazione in stringa.

Una *HashMap* associa ad ogni valore una stringa in modo da ottenere una corrispondenza biunivoca tra rappresentazione in stringa e valore dell’enumeratore, il tutto in tempo costante.

L’override del metodo *toString()* consente di ottenere un rappresentazione in stringa dell’enumereatore.

## Gestione dei path

I path della cartella in cui i file sono salvati vengono gestiti principalmente tramite la variabile *dataDirectory* nella classe *Data*. In questo modo risulta immediate cambiare tale cartella per tutti file in caso si decidesse di farlo. Per creare il path relativo base viene usata la variabile *class* di *Data*. Da questo si ottiene un URI che poi viene convertito in file. Tale file rappresenta il path assoluto del file eseguibile jar. Da qui si utilizza *getParentFile()* fino ad arrivare alla directory desiderata. Tutto ciò viene eseguito nell'inizializzatore statico per evitare conflitti infase di caricamento delle classi. Si noti come per i file venga usato il costruttore che ha come argomento un *File* parent oltre al nome.

## Mappatura a CSV

La maggior parte delle classi del package *data* implementano un metodo toRow() che restituisce una rappresentazione dell'oggetto come riga di un file CSV.
Analogamente un costruttore che ha come argomento un array di *String* crea un oggetto a partire da una riga di un file CSV nella quale ogni elemento rappresenta una cella.
In questo modo è possibile migliore la leggibilità del codice riducendo gli argomenti.
L'ordine dei dati nelle celle è specifico per ogni tipo di oggetto ed è specificato nella documentazione JavDoc.
In ogni caso la mappatura dei dati prevede l'inserimento dei valori associati alle variabili in numero fisso nella prima parte della riga, seguite da quelle in numero variabile (Eventi,Statistiche).

## Strutture dati

Per evitare di dover esaminare ogni carattere nei file ogni volta che si esegue un'operazonie si è deciso di caricare tutti i dati nell'applicazione all'avvio.

Questo fa anche in modo che l'utente debba aspettare il caricamento una sola volta e che le operazioni vengano eseguite velocemente.

Lo svantaggio è che il tempo di importazione potrebbe essere elevato e che si potrebbero creare problemi di utilizzo di memoria quando si ha a che fare con file di grandi dimensioni (esaurimento dello heap).

Per quanto riguarda il salvataggio dei dati si è deciso di scrivere i dati nei file qunado un' operazione che riguarda quel file viene eseguita. Ad eccezione delle operazioni su eventi avversi il salvataggio avviene aggiungendo i dati alla fine dei file riducendo così i tempi di salvataggio.

Le strutture dati più utilizzate sono le *HashMap* e le sue varianti *LinkedHashMap* e *LinkedListMultiMap*. Esse infatti consentono l'accesso a dati in tempo costante.

Per migliorare ulteriormente le prestazioni in fase di caricamento dei dati, si è deciso di creare un header in ogni file contenente il numero di oggetti che saranno contenuti nelle mappe.

Si evitano così inutili operazioni di rehashing creando una mappa delle giuste dimensioni.

Il risultato della ricerca dei centri è contenuto invece in una *LinkedList*.

## ID univoco

L'id univoco numerico assegnato ad ogni vaccinato è stato memorizzato in una variabile di tipo *long*.

Essendo necessario memorizzarne il valore di 16 cifre ($9^{16}$ possibili valori), l'utilizzo di un *long* consente di evitare l'overflow e di memorizzare un valore massimo di $2^{61}$-1.

## Eventi avversi

Gli eventi avversi sono memorizzati nel file nomeCentro_Vaccinati.csv (come da specifiche).

Al fine di organizzare meglio i dati ad ogni cittadino vaccinato sono associati i relativi eventi segnalati.

Ogni cittadino può segnalare un evento per tipo, ognuno dei quali può contenere un commento facoltativo.

Per aggiungere il supporto ad un nuovo tipo di evento avverso è sufficiente aggiungere un nuovo valore all'enumeratore *EventType*, poiché tutto ciò che riguarda gli eventi avversi è stato progettato tenendo presente l'estensibilità. Andrà modificata solo la parte relativa alle statistiche aggregate.

## Statistiche aggregate

Le statistiche aggregate sono memorizzate separatamente nel file *CentriVaccinali.csv*. In questo modo non è necessario calcolare la media iterando su tutti i dati dei cittadini vaccinati e le informazioni risultano meglio organizzate.

Come effetto collaterale è però necessario salvare nuovamente tutti i centri poiché non è possibile sovrascrivere una sola parte di file.

Le statistiche salvate comprendono una statistica globale e una statistica per ogni tipo di evento avverso, ognuna delle quali contiene il nome, il numero di segnalazioni e la media attuale.

Le statistiche di un centro vengono aggiornate e salvate quando un cittadino segnala un nuovo evento.

## Correttezza dei dati

La correttezza dei dati inseriti è delegata all'interfaccia grafica. Infatti ogni dato inserito viene controllato con un oggetto *Validator*.

La correttezza dei dati letti dai file è invece basata sull'assunzione che i file non vengano modificati manualmente dall'esterno del programma o che non vengano corrotti.

Essendo l'interfaccia grafica l'unico modo di inserire dati la loro correttezza è garantita anche in lettura.

## Gestione eccezioni

Le uniche eccezioni che vengono gestite tramite blocchi *try-catch* sono quelle controllate.

La maggior parte di esse vengono sollevate dalle classi del package *datamanager,* in quanto gestiscono la lettura e scrittura dei file.

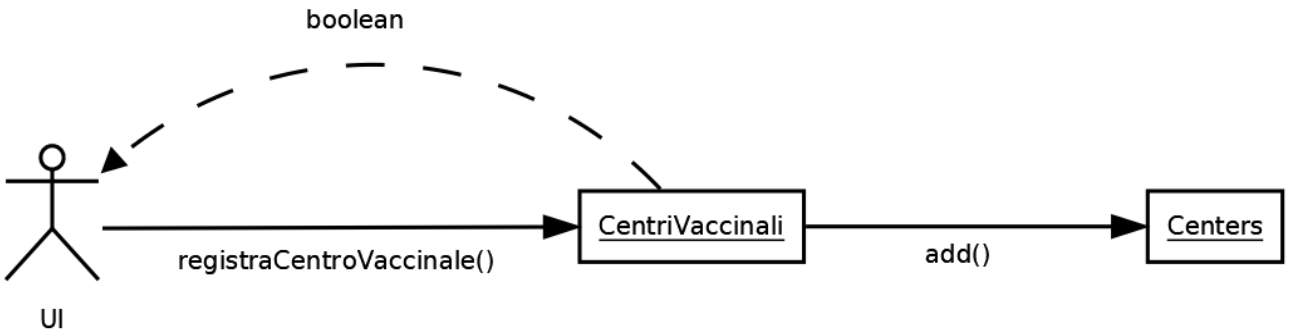
La gestione è affidata alle classi del package *ui*. In questo modo interfaccia e dati sono completamente indipendenti ed è inoltre possibile informare l'utente dell'errore e decidere come gestire l'eccezione.

Vengono anche utilizzati dei blocchi *try-with-resources* che consentono la corretta chiusura degli stream in caso di errori.

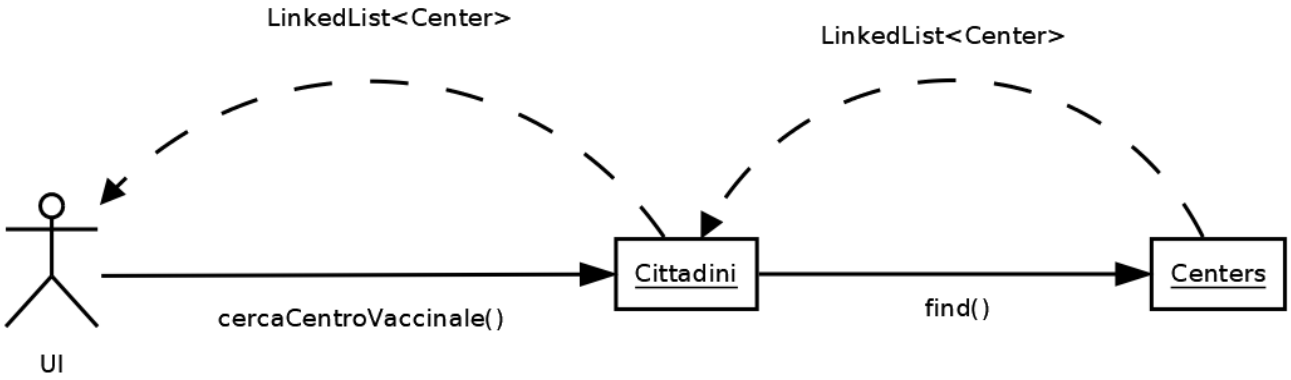
## Operazioni

Di seguito viene illustrato il funzionamento ad alto livello delle operazioni principali.

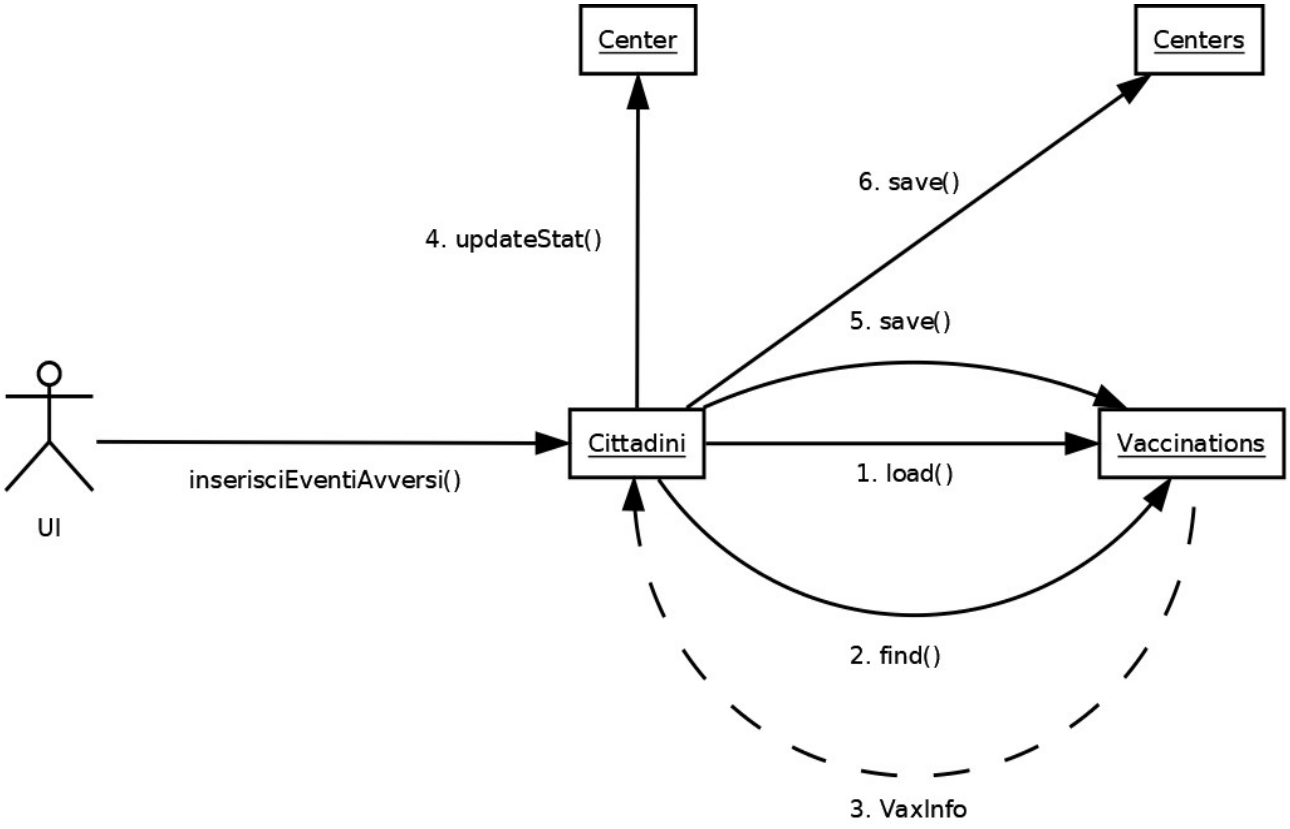
### Registrazione centro



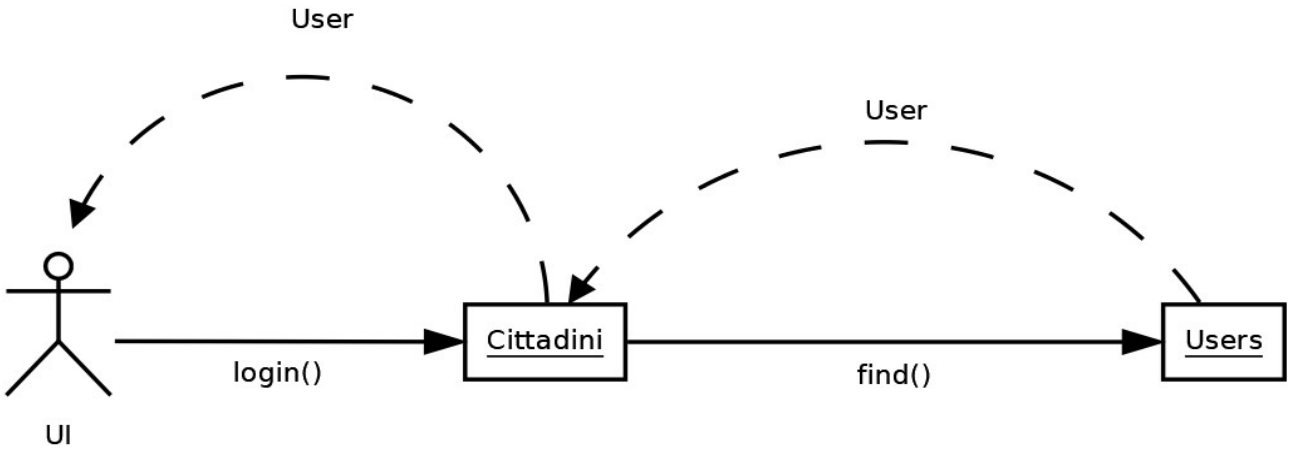

### Ricerca centro



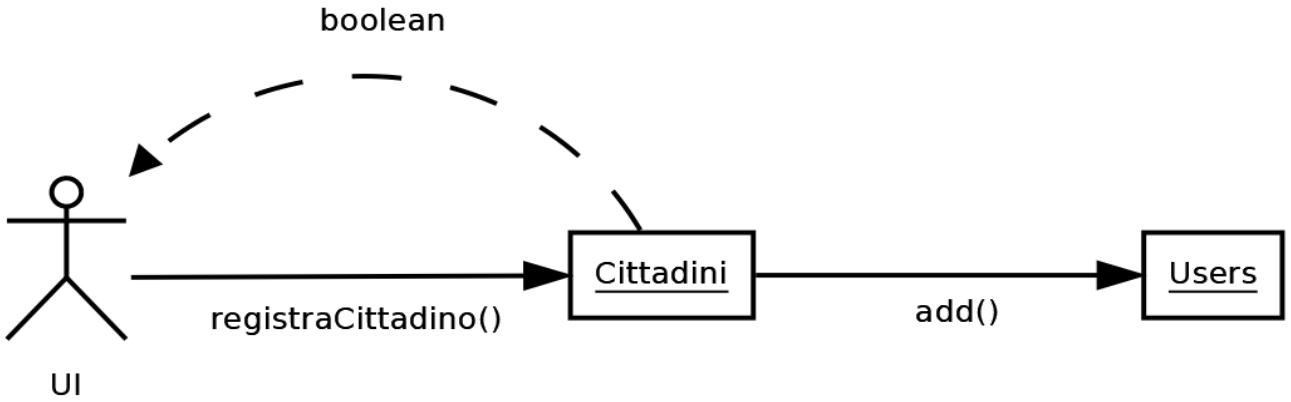

### Segnalazione evento



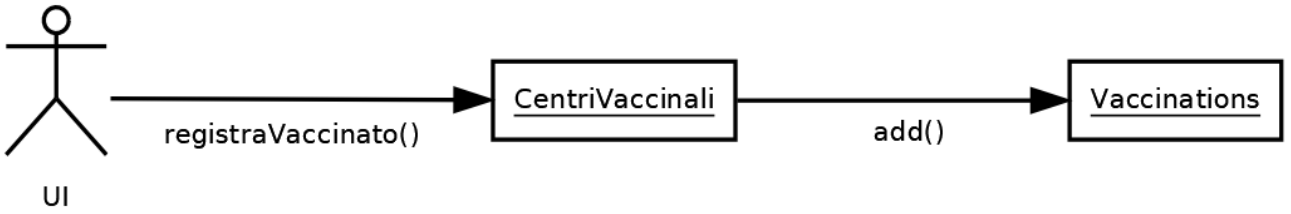
## Accesso cittadino




## Registrazione cittadino




## Registrazione vaccinato



# Packages

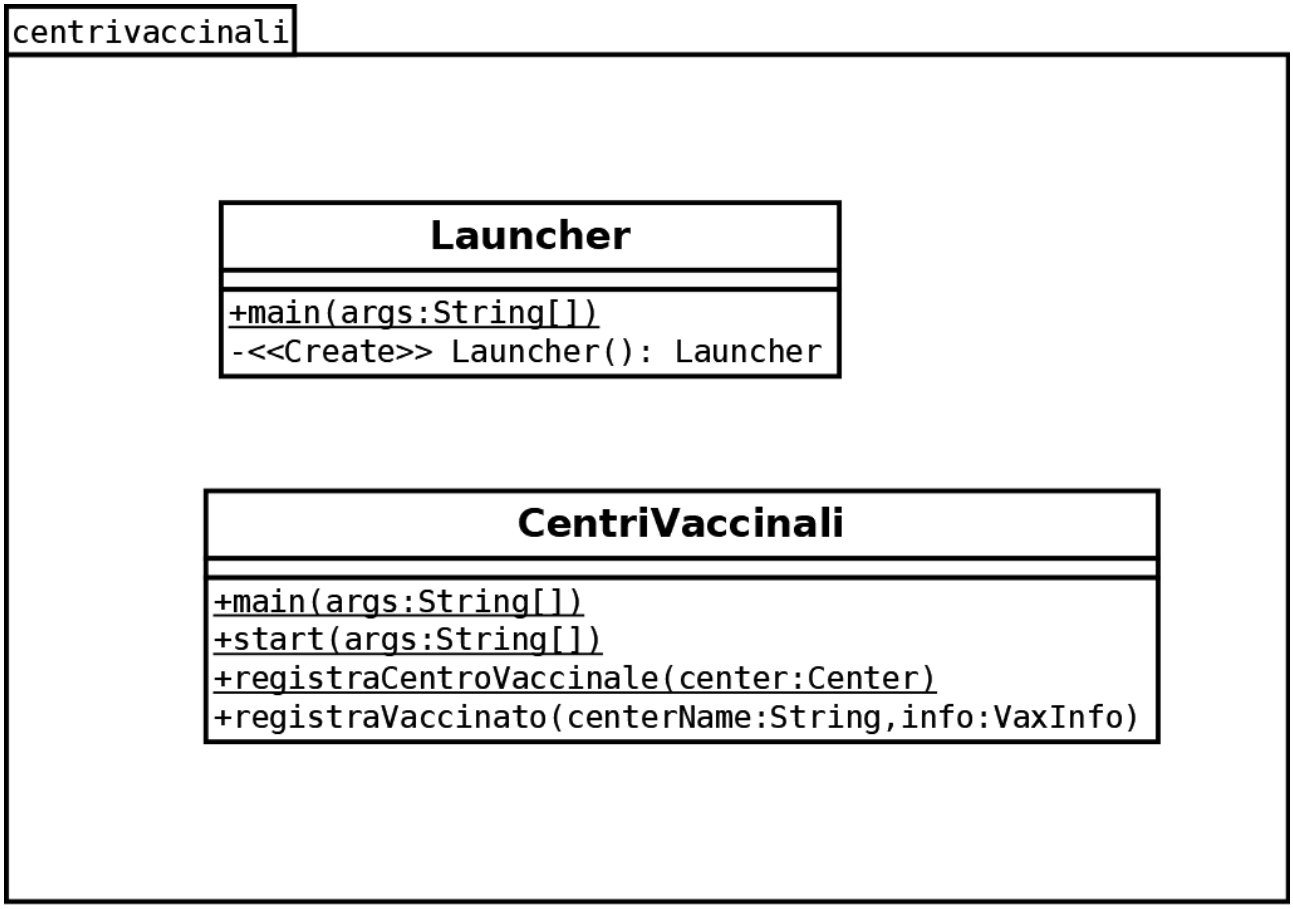
Il progetto è costituito da 5 packages: ui, data, datamanager, centrivaccinali e cittadini.

Di seguito saranno illustrati i principi base di funzionamento delle classi e di alcuni metodi che possono risultare complessi.

Per tutto ciò che non è riportato di seguito, si rimanda alla documentazione JavDoc.

## Centrivaccinali

Questo package richiesto dalle specifiche di progetto continene due classi: *Launcher* e *CentriVaccinali.*




### Launcher

Questa classe è stata aggiunta a causa di una incompatibilità tra moduli.

In particolare non è possibile implementare il metodo *main* in classi che estendono *Application* per poi creare un unico file *jar*.

La soluzione è invocare il metodo m*ain* di *CentriVaccinali* da questa classe.

Un costruttore privato previene la creazione di istanze.

### CentriVaccinali

Gestisce le operazioni e azioni della parte dei centri vaccinali.

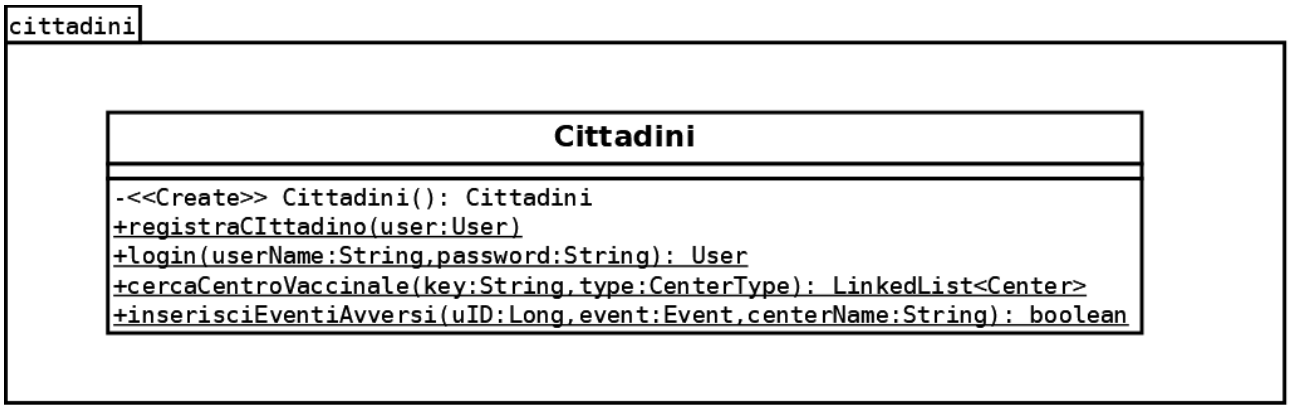
In particolare il metodo *main* (richiesto dalle specifiche) ha il compito di inizializzare l'interfaccia e caricare i dati dai file invocando il metodo *start*.

Il metodo registraVaccinato aggiunge un vaccinato mentre *registraCentroVaccinale* aggiunge un centro vaccinale.

Tutti i metodi sono statici in modo da poter essere invocati facilmente.

## Cittadini

Questo package richiesto dalle specifiche contiene la sola classe *Cittadini*.




### Cittadini

Gestisce le operazioni e azioni della parte dei cittadini.

Tutti i metodi sono statici in modo da poter essere invocati facilmente.

Un costruttore privato previene la creazione di istanze.

#### *cercaCentroVaccinale(String key, CenterType type)*

Ricerca un centro vaccinale per nome o comune e tipo.

Se *type* è *null*, viene eseguita una ricerca per nome, altrimenti per comune e tipo.

Sulla stringa *key* viene eseguita l'operazione *trim()* per rimuovere eventuali spazi.

Il risultato della ricerca è una *LinkedLIst* ordinata alfabeticamente utilizzando un *Comparator* che confronta i nomi dei centri.

La complessità di questa operazione è $\Theta(n \log(n))$ se i dati non sono parzialmente ordinati, poiché viene utilizzato un mergesort modificato.

### *login(String userName, String password)*

Esegue il login usando username e password.

Sulla stringa userName viene eseguita l'operazione *trim()* per eliminare eventuali spazi.

Il metodo ritorna l'oggetto *User* corrispondente agli argomenti che ha valore *null* se l'utente non è stato trovato o se la password è sbagliata.

### *inserisciEventiAvversi(Integer uID, Event e, Center center)*

Inserisce un evento avverso nei dati dell'utente associato all'id univoco uID.

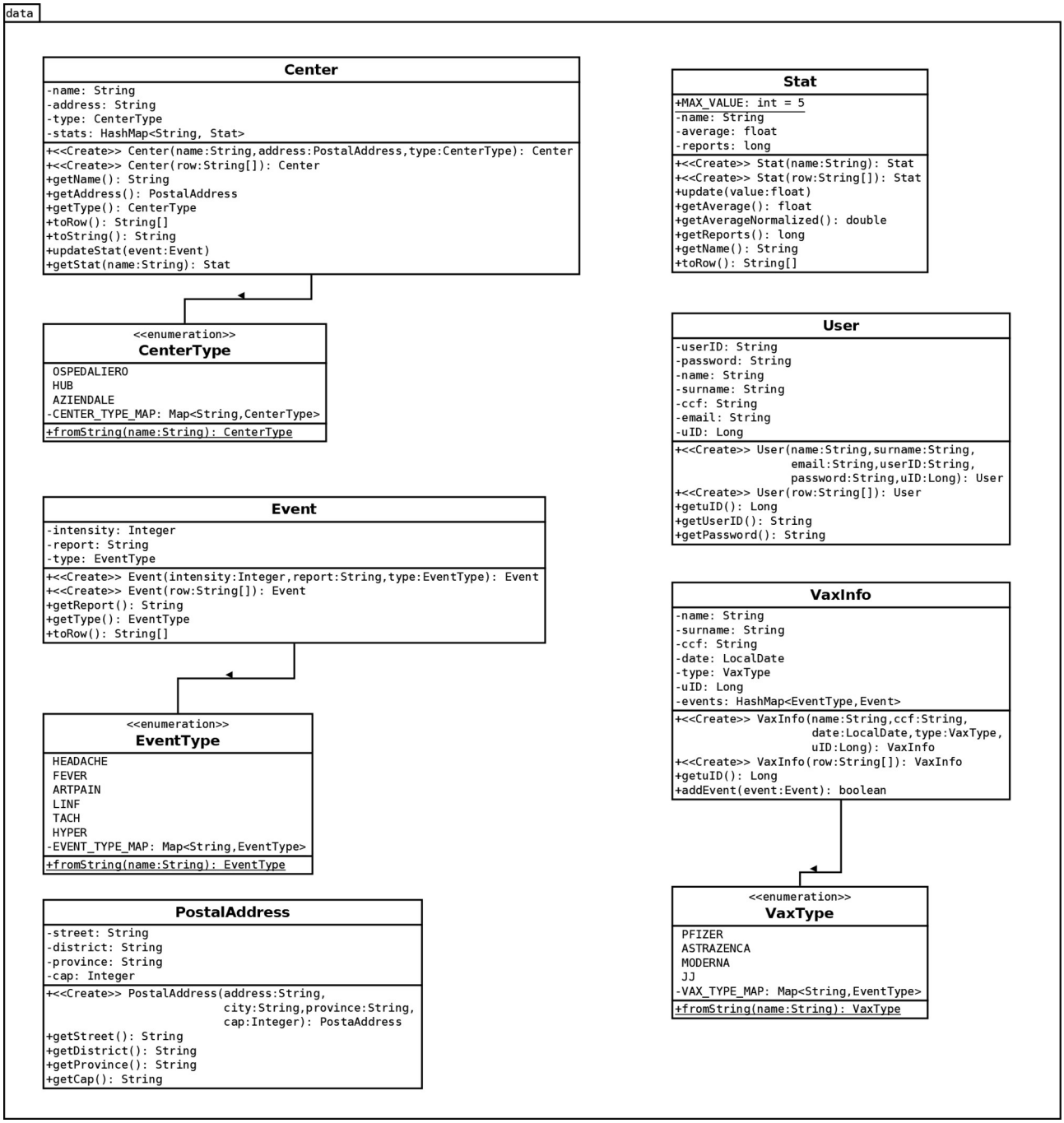
Viene sfruttato il fatto che l'oggetto *VaxInfo* nel quale sono contenuti gli eventi avversi sia un riferimento, evitando così di dover eliminare e reinserire un nuovo oggetto nella struttura dati sottostante.

Il valore ritornato è *false* se un evento dello stesso tipo è già stato segnalato oppure se l'id univoco non è stato trovato nel centro vaccinale.

Invocando questo metodo verranno aggiornate anche le statistiche globali e relative al tipo di evento nel centro *Center.*

# Data

Package contenente le classi rappresentanti oggetti e tipi enumerativi.




## Center

Classe che rappresenta un centro vaccinale e contenente le sue informazioni.

Si noti come sia stato eseguito l'override del metodo *toString()* in quanto viene utilizzato dall'interfaccia per visualizzare un prospetto riassuntivo del centro in seguito ad una ricerca, evitando di dover implementare una classe *ListCell* personalizzata.

Le statistiche aggregate sono contenute in una *HashMap* con chiave il nome della statistica e con valore un oggetto *Event*. In questo modo è possibile creare statistiche personalizzate oltre a quelle definite dall'enumeratore *EventType.*

### *updateStat(String name)*

Questo metodo ha il compito di aggiornare la statistica di nome *name*. Allo stesso tempo verrà aggiornata anche la statistica *Global* che rappresenta una media di tutte le altre.

## CenterType

Classe enumerativa che rappresenta il tipo di centro.

## Event

Classe che rappresenta un evento avverso e contenete tutti i dati necessari alla sua descrizione.

Ad ogni evento è associato un enumeratore di *EventType*.

## EventType

Classe enumerativa rappresentante il tipo di evento avverso.

## PostalAddress

Rappresenta un indirizzo postale e contiene via, cap, comune e provincia.

## User

Rappresenta un cittadino registrato.

## VaxInfo

Rappresenta le informazioni di vaccinazione di un cittadino vaccinato.

Oltre a contenere le informazioni contiene anche gli eventi avversi segnalati da un cittadino in una *HashMap*.

Durante la creazione della stessa è possibile calcolarne la dimensione esatta affinché non ne venga aumentata la dimensione suvessivamente.

Infatti il numero di tipi di eventi avversi è conosciuto e un cittadino può segnalare un solo evento per tipo.

Tutte le operazioni di aggiunta degli eventi avversi (escludendo caricamento e salvataggio dei file) hanno complessità in tempo O(1).

### *AddEvent(Event e)*

Aggiunge un evento avverso se e solo se un evento dello stesso tipo non è già presente.

Il valore ritornato è *true* se l'evento è aggiunto con successo, *false* altrimenti.

### *ToRow()*

Ritorna una rappresentazione dell'oggetto come riga di un file csv. In questo caso è possibile ottimizzare la lunghezza della riga evitando di inserire i tipi di eventi che non sono stati segnalati.

L'array contiene le prime sei variabili e successivamente sono inseritie le informazioni degli eventi in gruppi di tre elementi (le tre variabili della classe *Event*) usando il metodo *System.arraycopy()*.

### *VaxInfo(String[] row)*

Costruttore che crea un' istanza a partire da una riga di csv. Come nel metodo *toRow()* vengono assegnate prima le sei variabili e successivamente si creano gli oggetti event leggendo gli elementi dall'array in gruppi di tre con il metodo *Arrays.copyOfRange()*.

## VaxType

Classe enumerativa rappresentante il tipo di vaccino.

## Stat

Classe che rappresenta una statistica aggregata anonima. Ogni classe contiene il valore attuale della media di tutte le segnalazioni associate al tipo di statistica, il numero di segnalazioni, una stringa rappresentante il nome e una variabile statica *MAX_VALUE* che rappresenta il valora massima delle statistiche.

Questo consente di creare statistiche personalizzate oltre a quelle definite dal tipo enumerativo *EventType*.

Si è deciso di utilizzare il tipo *long* per la viariabile *reports* in modo da avere a disposizione un elevato numero di segnalazioni e il tipo *float* per la variabile *average* per ottenere un statistica più precisa.

### *update(float value)*

Aggiorna la statistica con un nuovo valore *value*, utilizzando la seguente formula:

$$newAverage = \frac{oldAverage * oldReports + newValue}{oldReports + 1}$$

### *getAverageNormalized()*

Ritorna il valore medio normalizzato, tramite la seguente formula:

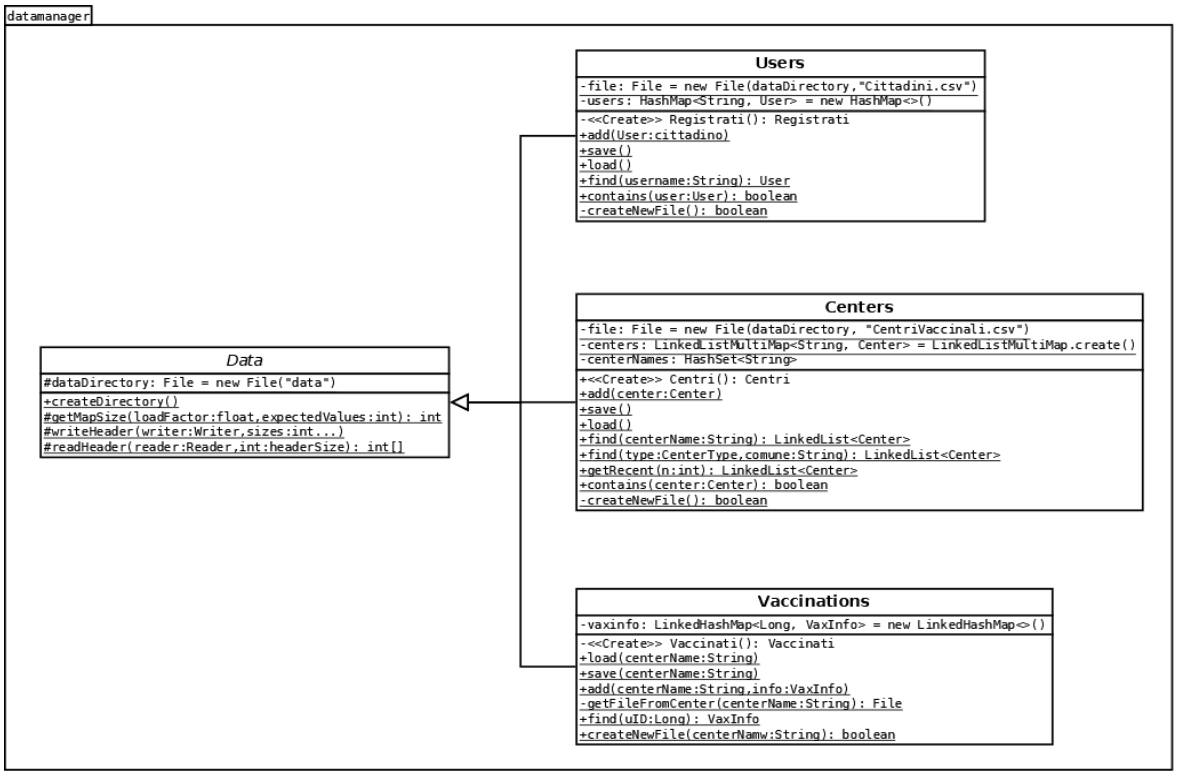
$$normAverage = \frac{average}{MAX\_VALUE}$$

Viene usato nell'interfaccia grafica.

# Datamanager

Questo package contiene classi per la gestione dei dati. Tutti i metodi di queste classi sono statici in quanto ogni classe rappresenta oggetti che non ammettono più di un'istanza. Inoltre così facendo è molto semplice invocare i metodi da qualsiasi classe.

Tutte le classi oltre a Vaccinations contengono una variabile file che rappresenta il file sul quale vengono eseguite le operazioni di lettura e scrittura.




## Data

La classe astratta data contiene variabili e metodi comuni alle classi del package.

In particolare la variabile *dataDirectory* contiene il path nel quale saranno salvati tutti file.

Questo permette inoltre di cambiare facilmente tale directory modificando solo questa variabile.

### *Headers*

Ogni file ha un header nella prima riga che contiene il numero di oggetti salvati nello stesso.

Il valore viene esteso per occupare esattamente 10 caratteri, aggiungendo "0" all'inizio. Ciò consente di sovrascrivere tale valore senza dover riscrivere tutta la porzione di file successiva.

Questo avviene quando si aggiungono nuovi elementi, mediante l'utilizzo di un *RandomAccessFile* che consente di modificare la posizione nella quale si vuole scrivere il valore tramite il metodo *seek()*.

È inoltre possibile saltare tale porzione di file quando si devono riscrivere tutti i dati, dopo aver invocato il metodo *setLength()* che imposta la lunghezza del file in byte.

Si noti come un carattere sia considerato come sequenza di 8 bit (1 byte).

Per separare l’header dal resto del file si è scelto il separatore “\n” in quanto migliora la formattazione del documento se aperto con programmi esterni. Non è da considerarsi come carattere di newline.

| HEADER (10 caratteri per intero) + SEPARATORE (\n) |
|---|
| RECORD 1 |
| RECORD 2 |
| ... |

Quando un file vuoto viene creato l’header è impostato a “0000000000\n”, ciò rappresenta il fatto che non ci sono dati.

### *CreateDirectory()*

Crea la directory nella quale saranno salvati i file se non esiste. Se la variabile *dataDirectory* è nulla, viene sollevata un’eccezione in quanto ciò significa che non è stato possibile creare tale cartella nell’inizializzatore statico.

### *getMapSize(float loadFactor, int expectedValues)*

Calcola la dimensione che una mappa deve avere affinché non vengano effettuate operazioni di rehashing inserendo i dati.

Viene usata la seguente formula:

$$capacity = \left\lceil \frac{expectedValues}{loadFactor} \right\rceil + 1$$

### *writeHeader(Writer writer, int… sizes)*

Questo metodo scrive un header utilizzando il *Writer* specificato.

Un ciclo for scrive una stringa rappresentante l’intero selezionato. Alla stringa vengono aggiunti degli “0” all’inizio fino a raggiungere una lunghezza di 10 caratteri.

In seguito viene scritto il separatore “\n” e successivamente viene invocato *writer.flush()*.

Ciò assicura che il contenunto di *writer* venga scritto immediatamente nel file.

### *readHeader(Reader reader, int headerSize)*

Questo metodo ha il compito di leggere un header e restituire i valori letti all’interno di un array di interi, nell’ordine in cui compaiono nel file.

L’argomento *headerSize* rappresenta il numero di interi che l’header contiene.

Tramite *reader.read()* si leggono 10 caratteri alla volta in un buffer, che poi verrà convertito in intero creando una nuova stringa e usando I*nteger.parseInt()*.

In conclusione viene eseguito *reader.skip(1)* per avanzare il *reader* di 1 carattere, ossia il separatore “\n”.

## Centers

Classe che gestisce il file CentriVaccinali.csv.I dati letti sono inseriti in un LinkedListMultiMap che consente di avere più valori per la stessa chiave, usando come struttura dati sottostante una LinkedHashMap. Ciò consente inoltre di mantenere l'ordine di inserimento dei valori, che rispecchia quello di registrazione dei centri. Come chiave è stato usato il nome del comune (minuscolo) nel quale un determinato centro si trova. Un ulteriore HashSet contenente i nomi dei centri viene usato per controllare l'esistenza del centro in fase di registrazione. Quando è necessario scrivere o leggere dati il FileReader o FileWriter viene "wrappato" in un BufferedReader o BufferedWriter per migliorare le prestazioni.

### *Load()*

Questo metodo carica nella *MultiMap* e nell'*HashSet* tutti i dati dei centri vaccinali dal file *CentriVaccinali.csv*, dopo aver creato la mappa e il set delle giuste dimensioni.

Prima però si controlla l'esistenza del suddetto file e lo si crea nel caso in cui non esista.

Vengono anche creati i file relativi ai vaccinati mediante il metodo Vaccinations.createFile().

Per leggere i dati viene usato un *Iterator <String[]>* creato grazie alla classe *CsvParser*.

L'iteratore consente di iterare le righe del file csv come un normale iteratore.

### *Save()*

Salva tutti i centri attualmente contenuti nella mappa *centers*.

Dopo aver reimpostato la dimensione del file a 11 byte (header+separatore), si scrivono tutti i centri con un *CsvWriter*.

### *Add()*

Viene aggiunto un nuovo centro alla fine del file, dopo aver aggiornato l'header.

In questo particolare caso le prime 10 cifre dell'header rappresentano il numero di centri mentre le 10 successive il numero di comuni.

Grazie a questi valori si può calcolare la dimensione delle mappe *centerNames* e *centers*.

Si noti come grazie al *RandomAccessFile* sia possibile saltare direttamente alla fine del file dopo l'aggiornamento dell'header, senza riscrivere tutti i centri.

Anche qui viene usato un *CsvWriter*.

### *Find(String centerName)*

Ricerca un centro per nome.

Il valore ritornato è una *LinkedList* al cui interno si trovano tutti i centri il cui nome contiene la stringa *centerName*. La comparazione avviene su stringhe in minuscolo, così facendo la ricerca è case-insensitive.

La lista è vuota se nessun centro è stato trovato.

L’operazione ha complessità Θ(n) con n numero di centri, poiché essi devono essere attraversati tutti per ricercare la stringa.

### *Find(CenterType type, String comune)*

Ricerca un centro per tipo e comune.

Il valore ritornato è una *LinkedList* al cui interno si trovano tutti i centri nel comune *comune* e di tipo *type*. La stringa *comune* è convertita in minuscolo, così facendo la ricerca è case-insensitive.

La lista è vuota se nessun centro è stato trovato.

L’operazione ha complessità Θ(m) con m numero di centri in *comune* perché oltre ad ottenere una lista di centri nel comune (O(1)), bisogna anche controllare il tipo di ogni centro ed eventualmente aggiungerlo alla lista.

## Users

La classe gestisce i dati dei cittadini registrati, contenuti nel file *Cittadini.csv*.

I dati sono inseriti in una *HashMap* che ha come chiave il nome utente (minuscolo) del registrato.

L’header contiene un solo valore.

### *Add(User user)*

Funziona in modo simile a *Centers.add()*.

### *Load()*

Questo metodo carica nella *HashMap* tutti i dati degli utenti dal file *Cittadini.csv*.

Allo stesso tempo si controlla l’esistenza del suddetto file e lo si crea nel caso in cui non esista.

Funziona in modo simile al metodo *Centers.load()*.

### *Find(String username)*

Questo metodo esegue una ricerca per nome utente.

In questo caso la ricerca è case-sensitive.

Viene ritornato un oggetto *User* oppure *null* se l’utente non esiste.

La complessità dell’operazione è O(1) in quanto l’accesso ad una *HashMap* ha una complessità costante.

## Vaccinations

Gestisce tutti i file dei cittadini vaccinati e i corrispondenti eventi avversi. I file, come da specifiche di progetto, hanno nome che segue la convenzione NomeCentro_Vaccinati.csv. I dati relativi ad un determinato centro sono caricati all’interno dalla LinkedHashMap vaxinfo. Le successive operazioni (ricerca, salvataggio), saranno effettuate su questi dati. Si è deciso di utilizzare questa struttura per ottenere prestazioni migliori quando è necessario salvare tutte le informazione dei vaccinati, che è una delle operazioni che richiede più

tempo. Infatti questa struttura dati consente di iterare gli elementi con un tempo proporzionale alla dimensione della mappa e non alla sua capacità. L'header di ogni file contiene un solo valore.

### *add(String centerName, VaxInfo info)*

Aggiunge un vaccinato alla fine del file.

Funziona in modo simile a *Centers.add()* ma in questo caso bisogna aggiornare l'header poiché non è necessario caricare tutti i dati nella mappa.

Per farlo si usa il metodo *readHeader()* e successivamente si scrive nel file con il metodo *writeHeader()*.

### *Load(String centerName)*

Legge ed inserisce nella *LinkedHashMap* tutti i dati dei vaccinati relativi al centro *centerName*.

Funziona in modo simile a *Centers.load()*.

### *getFileFromCenter(String centerName)*

Questo metodo ha il compito di creare un oggetto *File* relativo al centro *centerName*, componendo il nome del file come da convenzione.

### *Save(String centerName)*

Scrive nel file relative a *centerName* tutti i file attualmente contenuti nella LinkedHashMap.

Questa operazione è necessaria in quanto non è possibile sovrascrivere i vecchi dati in seguito all'aggiunta di un evento avverso poiché le righe contengono celle con dimensione variabile.

È quindi necessario salvare nuovamente le informazioni.

Funziona in modo simile a *Centri.save()*.

### *find(long uID)*

Ritorna un oggetto *VaxInfo* relativo ad un cittadino con id univoco *uID*. Viene ritornato *null* se l'id univoco non esiste.

La complessità di questa operazione è O(1) in quanto si esegue un accesso ad una *LinkedHashMap*.

## Ui

Nonostante le specifiche non richiedano l'implementazione di un'interfaccia grafica, si è deciso di crearne una per avvicinarsi il più possibile alle esigenze dell'utente.

È stato scelto il tema Material Design in quanto ben documentato.

Le classi di questo package si dividono in tre categorie: gestione pagine, pagine, componenti e layout riutilizzabili.

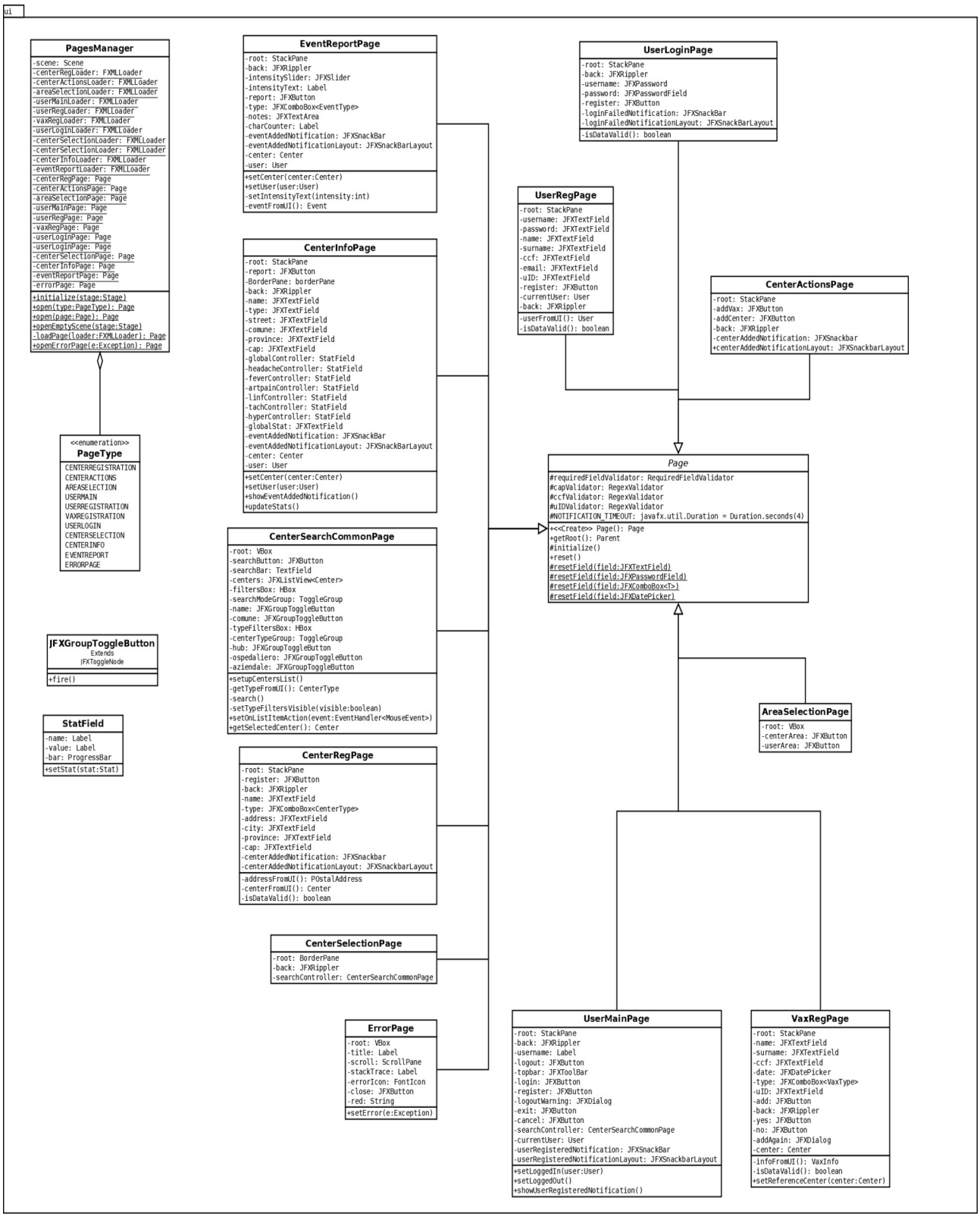
L'interfaccia ha inoltre il compito di verificare la correttezza dei dati inseriti e risulta modulare, nel senso che sarebbe possibile cambiare la sua implementazione (librerie diverse, terminale, ecc.) senza dover effettuare alcun cambiamento alle altre classi (oltre a *CentriVaccinali* dove viene inizializzata l'interfaccia).

Le diverse schermate sono rappresentate da classi che estendono la classe astratta Page, che fungono anche da controller per i layout FXML.

Si è deciso di utilizzare il più possibile i file di layout FXML per ottenere del codice più pulito e di facile lettura.

L'unica eccezione è la classe *ErrorPage*, il cui layout viene creato invocando il metodo *initialize()*.

Questa decisione è stata presa per fare in modo che questa pagina fosse sempre disponibile, anche nel caso in cui i file FXML non possano essere caricati.


ui
PagesManager
-scene: Scene
-centerRegLoader: FXMLLoader
-centerActionsLoader: FXMLLoader
-areaSelectionLoader: FXMLLoader
-userMainLoader: FXMLLoader
-userRegLoader: FXMLLoader
-vaxRegLoader: FXMLLoader
-userLoginLoader: FXMLLoader
-centerSelectionLoader: FXMLLoader
-centerSelectionLoader: FXMLLoader
-centerInfoLoader: FXMLLoader
-eventReportLoader: FXMLLoader
-centerRegPage: Page
-centerActionsPage: Page
-areaSelectionPage: Page
-userMainPage: Page
-userRegPage: Page
-vaxRegPage: Page
-userLoginPage: Page
-userLoginPage: Page
-centerSelectionPage: Page
-centerInfoPage: Page
-eventReportPage: Page
-errorPage: Page
+initialize(stage:Stage)
+open(type:PageType): Page
+open(page:Page): Page
+openEmptyScene(stage:Stage)
-loadPage(loader:FXMLLoader): Page
+openErrorPage(e:Exception): Page
<<enumeration>>
PageType
CENTERREGISTRATION
CENTERACTIONS
AREASELECTION
USERMAIN
USERREGISTRATION
VAXREGISTRATION
USERLOGIN
CENTERSELECTION
CENTERINFO
EVENTREPORT
ERRORPAGE
JFXGroupToggleButton
Extends
JFXToggleNode
+fire()
StatField
-name: Label
-value: Label
-bar: ProgressBar
+setStat(stat:Stat)
EventReportPage
-root: StackPane
-back: JFXRippler
-intensitySlider: JFXSlider
-intensityText: Label
-report: JFXButton
-type: JFXComboBox<EventType>
-notes: JFXTextArea
-charCounter: Label
-eventAddedNotification: JFXSnackBar
-eventAddedNotificationLayout: JFXSnackBarLayout
-center: Center
-user: User
+setCenter(center:Center)
+setUser(user:User)
-setIntensityText(intensity:int)
-eventFromUI(): Event
CenterInfoPage
-root: StackPane
-report: JFXButton
-BorderPane: borderPane
-back: JFXRippler
-name: JFXTextField
-type: JFXTextField
-street: JFXTextField
-comune: JFXTextField
-province: JFXTextField
-cap: JFXTextField
-globalController: StatField
-headacheController: StatField
-feverController: StatField
-artpainController: StatField
-linfController: StatField
-tachController: StatField
-hyperController: StatField
-globalStat: JFXTextField
-eventAddedNotification: JFXSnackBar
-eventAddedNotificationLayout: JFXSnackBarLayout
-center: Center
-user: User
+setCenter(center:Center)
+setUser(user:User)
+showEventAddedNotification()
+updateStats()
CenterSearchCommonPage
-root: VBox
-searchButton: JFXButton
-searchBar: TextField
-centers: JFXListView<Center>
-filtersBox: HBox
-searchModeGroup: ToggleGroup
-name: JFXGroupToggleButton
-comune: JFXGroupToggleButton
-typeFiltersBox: HBox
-centerTypeGroup: ToggleGroup
-hub: JFXGroupToggleButton
-ospedaliero: JFXGroupToggleButton
-aziendale: JFXGroupToggleButton
+setupCentersList()
-getTypeFromUI(): CenterType
-search()
-setTypeFiltersVisible(visible:boolean)
+setOnListItemAction(event:EventHandler<MouseEvent>)
+getSelectedCenter(): Center
CenterRegPage
-root: StackPane
-register: JFXButton
-back: JFXRippler
-name: JFXTextField
-type: JFXComboBox<CenterType>
-address: JFXTextField
-city: JFXTextField
-province: JFXTextField
-cap: JFXTextField
-centerAddedNotification: JFXSnackbar
-centerAddedNotificationLayout: JFXSnackbarLayout
-addressFromUI(): POstalAddress
-centerFromUI(): Center
-isDataValid(): boolean
CenterSelectionPage
-root: BorderPane
-back: JFXRippler
-searchController: CenterSearchCommonPage
ErrorPage
-root: VBox
-title: Label
-scroll: ScrollPane
-stackTrace: Label
-errorIcon: FontIcon
-close: JFXButton
-red: String
+setError(e:Exception)
UserLoginPage
-root: StackPane
-back: JFXRippler
-username: JFXPassword
-password: JFXPasswordField
-register: JFXButton
-loginFailedNotification: JFXSnackBar
-loginFailedNotificationLayout: JFXSnackBarLayout
-isDataValid(): boolean
UserRegPage
-root: StackPane
-username: JFXTextField
-password: JFXTextField
-name: JFXTextField
-surname: JFXTextField
-ccf: JFXTextField
-email: JFXTextField
-uID: JFXTextField
-register: JFXButton
-currentUser: User
-back: JFXRippler
-userFromUI(): User
-isDataValid(): boolean
CenterActionsPage
-root: StackPane
-addVax: JFXButton
-addCenter: JFXButton
-back: JFXRippler
-centerAddedNotification: JFXSnackbar
+centerAddedNotificationLayout: JFXSnackbarLayout
Page
#requiredFieldValidator: RequiredFieldValidator
#capValidator: RegexValidator
#ccfValidator: RegexValidator
#uIDValidator: RegexValidator
#NOTIFICATION_TIMEOUT: javafx.util.Duration = Duration.seconds(4)
+<<Create>> Page(): Page
+getRoot(): Parent
#initialize()
+reset()
#resetField(field:JFXTextField)
#resetField(field:JFXPasswordField)
#resetField(field:JFXComboBox<T>)
#resetField(field:JFXDatePicker)
AreaSelectionPage
-root: VBox
-centerArea: JFXButton
-userArea: JFXButton
UserMainPage
-root: StackPane
-back: JFXRippler
-username: Label
-logout: JFXButton
-topbar: JFXToolBar
-login: JFXButton
-register: JFXButton
-logoutWarning: JFXDialog
-exit: JFXButton
-cancel: JFXButton
-searchController: CenterSearchCommonPage
-currentUser: User
-userRegisteredNotification: JFXSnackBar
-userRegisteredNotificationLayout: JFXSnackbarLayout
+setLoggedIn(user:User)
+setLoggedOut()
+showUserRegisteredNotification()
VaxRegPage
-root: StackPane
-name: JFXTextField
-surname: JFXTextField
-ccf: JFXTextField
-date: JFXDatePicker
-type: JFXComboBox<VaxType>
-uID: JFXTextField
-add: JFXButton
-back: JFXRippler
-yes: JFXButton
-no: JFXButton
-addAgain: JFXDialog
-center: Center
-infoFromUI(): VaxInfo
-isDataValid(): boolean
+setReferenceCenter(center:Center)

## PagesManager

Questa classe si occupa della navigazione tra pagine, ispirandosi ad una factory.

Le variabili rappresentano gli *FXMLLoader,* il cui compito è di caricare il layout memorizzato nei corrispondenti file, e le pagine.

Si segue la convenzione *pageNameLoader* per i loader e *pageNamePage* per le pagine.

In questo modo è possibile riutilizzare le pagine riducendo i tempi di caricamento e l'utilizzo di risorse.

Lo svantaggio è che le pagine vanno reimpostate al loro stato iniziale manualmente se necessario.

### *loadPage(FXMLLoader loader)*

Si occupa di caricare i controller associati ad un FXMLLoader. Se non è possibile caricare la pagina viene ritornato *null*  stampato lo stacktrace della *IOException*.

### *Initialize()*

Si occupa di caricare i controller associati agli FXMLLoaders.

### *Open(PageType type)*

Apre la pagina *Page*. Il meccanismo di apertura consiste nel sostituire in nodo radice corrente con quello della pagina che si vuole aprire.

Viene mostrata una schermata di errore se il metodo non ha implementato il tipo di pagina richiesto.

In caso di apertura riuscita viene restituito un riferimento *Page* alla pagina aperta.

### *Open(Page page)*

Si occupa di sostituire il nodo radice corrente con quello della nuova pagina.

### *OpenEmptyScene()*

Crea una nuova scena con una dimensione di 1280x720 px. Tale dimensione è sufficiente a visualizzare tutte le pagine correttamente.

Il nodo usato è un *Hbox* vuoto.

### *PageType*

Enumeratore innestato che rappresenta il tipo di pagina.

### *OpeErrorPage()*

È un metodo di convenienza usato per aprire una pagina di errore passando un'eccezione come argomento.

## Page

Classe astratta che rappresenta una pagina e che tutte le classi che rappresentano una pagina devono estendere.

Contengono alcune variabili comuni alla maggior parte delle pagine tra cui oggetti *Validator*, il cui scopo è di controllare la correttezza dei dati inseriti nell'interfaccia, e la variabile *NOTIFICATION_DURATION* che rappresenta il tempo in secondi dopo il quale una notifica viene nascosta.

Il costruttore di questa classe viene chiamato da tutte le sottoclassi e imposta le espressioni regolari dei *Validator*s che lo richiedono.

### *ccfValidator*

Istanza di *RegexValidator* che ha il compito di verificare la correttezza del codice inserito.

Viene utilizzata l'espressione regolare:

^[A-Za-z]{6}[A-Za-z0-9]{3}[0-9]{2}[A-Za-z0-9]{4}[A-Za-z]{1}$

Essa viene compilata per migliorare le prestazioni e controlla che:

- i primi sei caratteri siano lettere ( *[A-Z]{6}* )
- i tre caratteri successivi siano alfanumerici ( *[A-Za-z0-9]{3}* )
- i due caratteri successivi siano numerici ( *[0-9]{2}* )
- i successivi quattro caratteri siano alfanumerici ( *[A-Za-z0-9]{4}* )
- l'ultimo carattere sia una lettera ( *[A-Za-z]{1}$* )

### *capValidator*

Isatnza di *RegexValidator* che controlla la correttezza del CAP.

Viene utilizzata l'espressione regolare:

"^[0-9]{5}$"

Essa viene precompilata per migliorarne le prestazione e controlla che la stringa contenga 5 caratteri numerici.

### *resetField()*

Il metodo statico *resetField* il cui compito è quello di reimpostare un campo al suo stato di default. Si è deciso di eseguire l'overloading poiché ogni campo è leggermente diverso dagli altri e richiede operazioni diverse.

### *reset()*

Metodo atratto che permette di reimpostare la pagina al suo stato originale. Deve esserne eseguito l'override in ogni pagina.

### *GetRoot()*

Ritorna il nodo radice della pagina che verrà usato per sostituire la pagina precedente. Poiché ogni pagina ha un nodo radice di tipo diverso è necessario eseguire l'override di questo metodo.

### *Initialize()*

Questo metodo viene invocato automaticamente quando un file FXML viene caricato.

## Sottoclassi Page

Le classi che estendono *Page* rappresentano pagine diverse ma funzionano tutte in maniera simile. Nel metodo *initialize()* viene useguito un setup iniziale della pagina e vengono assegnati i metodi da eseguire quando si interagisce con l'interfaccia (bottoni, ecc.) per lo più tramite espressioni lambda, in modo da migliorare la leggibilità del codice.

Queste espressioni invocano anche metodi delle classi *Cittadini* e *CentriVaccinali*.

A volte viene anche richiamato il metodo reset e vengono nascosti o visualizzati elementi dinamici (bottoni di login, bottone di segnalazione evento avverso, ecc.).

Tranne alcuni casi particolari, le pagine vengono resettate quando si apre una nuova pagina.

Alcune pagine prevedono il passaggio di dati tra loro, che viene eseguito sfruttando metodi specifici e cast dei valori ritornati da *PagesManger.open(PageType type)*.

### *ErrorPage*

Questa classe rappresenta una pagina di errore. Il suo ruolo è di informare l'utente riguardo ad un errore che si è verificato.

Oltre ad un *Label* il cui scopo è di mostrare uno *stackTrace* dell'eccezione, è presente anche un *JFXButton* che consente all'utente di uscire dall'applicazione in maniera controllata.

### *CenterSearchCommonPage*

Questa classe particolare contiene l'interfaccia dedicata alla ricerca dei centri.

Essa non è una vera e propria pagina ma le è assimilabile per le sue funzionalità.

Questa pagina è riutilizzabile e viene infatti aggiunta alle classi *CenterSelectionPage*, nell'ambito di registrazione di un vaccinato per facilitare la selezione del centro, e *UserMainPage*.

Questo permette di ridurre la complessità delle suddette classi e migliorarne la leggibilità.

## Componenti e layout riutilizzabili

Si è reso necessario implementare alcuni componenti e layout riutilizzabili.

### *JFXGroupToggleButton*

Classe che estende *JFXToggleNode*. Implementa un *ToggleGroup* che permette la selezione di un solo bottone alla volta, quando il gruppo ne contiene più di uno. Viene utilizzata per la gestione dei filtri di ricerca dei centri.

### *StatField*

Controller del layout riutilizzabile che consente la visualizzazione di una statistica aggregata. Contiene due *Label*, uno per visualizzare il valore della statistica e uno per visualizzarne il nome, e

una *ProgressBar.* Contiene il solo metodo *setStat(Stat stat)* che permette di impostare i valori e il nome della statistica aggregata *stat*.

## Complessità operazioni

Di seguito viene riportata una tabella della complessità approssimativa di ciascuna operazione, considerando la lettura e scrittura dei file come operazioni aventi complessità costante, numero di statistiche ed eventi avversi costante e ricerca di una stringa in un'altra costante.

| OPERAZIONE | COMPLESSITÀ | VARIABILI |
|---|---|---|
| Ricerca centro (comune & tipo) | $\Theta(m)$ | m = centri nel comune cercato |
| Ricerca centro (nome) | $\Theta(n)$ | n = centri registrati |
| Registrazione centro | O(1) | |
| Registrazione cittadino | O(1) | |
| Registrazione vaccinato | O(1) | |
| Segnalazione evento avverso | $\Theta(n + m)$ | n = vaccinati nel centro<br>m = centri registrati |
| Calcolo statistiche centro | O(1) | |
| Accesso cittadino | O(1) | |

# LIMITAZIONI

La limitazione principale dell'applicazione è la scalabilità. Caricando in fatti in memoria tutti i dati, il consumo di memoria risulta ecceessivo quando si considerano file di grandi dimensioni, in particolare "Cittadini.csv" e "nomeCentro_Vaccinati.csv".

Lo stesso avverrebbe se non si caricassero in memoria tutti i dati ma si eseguisse la scansione degli stessi di volta in volta. In questo caso però tutte le operazioni risulterebbero rallentate con un'esperienza da parte dell'utente scadente.

L'applicazione inoltre supporta solo caratteri UTF-8 per questioni di compatibilità multi piattaforma e per ridurre le dimensioni dei file.

# POSSIBILI MIGLIORAMENTI

## Scalabilità

Per alleviare il problema della scalabilità si è considerato di poter caricare in memoria solo i dati necessari alle operazioni, memorizzati in file di indice. In essi sarebbero memorizzati una chiave da utilizzare nelle mappe associata ad un puntatore (*long*) ad una posizione nei file contenenti i dati.

I dati verrebbero prelevati da questi ultimi mediante un *RandomAccessFile* utilizzando il metodo *seek()*.

Il file di indice verrebbe aggiornato quando si aggiungono nuovi dati e permetterebbe una segnalazione degli eventi molto più veloce.

Purtroppo a causa di vincoli di tempo non è stato possibile implementare questa soluzione.

## Lettura e scrittura oggetti

I metodi di lettura e scrittura degli oggetti nelle classi *datamanger* potrebbero essere unificati in un solo metodo avente come argomento una *Map* e implementando un'interfaccia CSVMapped.

Tale interfaccia conterrebbe tre metodi: *getKey()*, *toRow()* e *fromRow()*.

Implementando questa interfaccia gli oggetti potrebbero essere creati da una riga di csv ed essere inseriti in una mappa di oggetti CSVMapped avente come chiave l'oggetto ritornato da *getKey()*.

Una factory di CSVMapped provvederebbe alla creazione di oggetti durante la lettura chiamando il metodo *fromRow()* su un oggetto vuoto.

Purtroppo non è stato possibile implementare questa soluzione a causa di un mancanza di tempo.

# BIBLIOGRAFIA


CSV peformance comparison – SimpleFlatMapper

List.sort() complexity - Java 16 API

Factory pattern – tutorialspoint.com

LinkedListMultiMap – Guava API

Enum pattern - Joshua Bloch Effective Java

InputSreamReader - Java 16 API

OutputStreamWriter - Java 16 API

Material Design - material.io