POLITECNICO DI TORINO

Dipartimento di Ingegneria Gestionale e della Produzione
Corso di Laurea in Ingegneria Gestionale
Classe L-8 Ingegneria dell'Informazione

Centrale Telefonica

Relatore
Prof. Fulvio Corno

Candidato
Marco Clemente Nanci (s204471)

Anno accademico
2017/2018

Indice

Proposta

Studente proponente

Marco Clemente Nanci (s204471)

Descrizione del problema proposto

Il problema consiste nel dover gestire i movimenti degli operatori della centrale telefonica al fine da permettere all'azienda di massimizzare il numero di operazioni effettuate sul territorio nell'arco di un determinato periodo di tempo(1 giorno,1 settimana).

Descrizione della rilevanza gestionale del problema

L'organizzazione delle mansioni da affidare agli operatori ricopre una notevole rilevanza gestionale , una buona strategia permette infatti di raggiungere gli obbiettivi prefissati che si traducono in ricavi per l'azienda. Nei vari ambiti dell'organizzazione interna i trasporti ricoprono certamente un ruolo fondamentale perché il tempo è una risorsa limitata, quindi la corretta gestione degli stessi permette all'azienda di ottenere ricavi utili e di guadagnarci in immagine nei confronti dei clienti.

Descrizione dei data-set per la valutazione

I dati verranno prelevati da una famosa azienda telefonica. Il database sarà formato da una tabella riguardante le centrali e una riguardante le operazioni. I dati sono semplificati perché le operazioni fanno riferimento a diversi interventi sul territorio(installazioni, riparazioni, guasti...) e il tempo medio è un tempo stimato, ma realistico. La lista delle centrali non è completa quindi lavorerò ipotizzando che esistano soltanto una parte di esse. Anche il numero degli operatori è reale anche se nell'applicazione si ipotizzerà che tutti siano in grado di portare a termine tutte le tipologie di operazioni da compiere sul territorio. I dati fanno riferimento all'anno 2016/2017 e ovviamente verranno omessi dati riguardanti operazioni ancora da portare a termine. Le coordinate geografiche degli indirizzi verranno forniti tramite API google maps

Descrizione preliminare degli algoritmi coinvolti

Per quanto riguarda la parte algoritmica l'obiettivo sarebbe quello di calcolare il percorso che gli operatori devono compiere per poter massimizzare le operazioni portate a termine nell'arco di tempo stabilito. Verrà creato un grafo pesato che collegherà le centrali alle operazioni, ipotizzando che le operazioni vengano smistate verso la centrale più vicina, e dove il peso dell'arco corrisponde al tempo necessario per arrivare sul posto e portare a termine l'operazione.

Descrizione preliminare delle funzionalità previste per l'applicazione software

L'applicazione permetterà di aggiungere un'operazione, una centrale, calcolare e visualizzare il percorso migliore per massimizzare il numero di operazioni compiute nell'arco di tempo stabilito.Verrà fornito un simulatore che permetterà di visualizzare il percorso degli operatori.

Descrizione dettagliata del problema

Il problema affrontato riguarda la gestione e il monitoraggio di un'impresa e l'ottimizzazione dei tempi di processo al fine di migliorare le prestazioni della suddetta. Il monitoraggio può intendersi come il complesso di attività volte a verificare la corretta attuazione dell'impostazione strategica adottata dall'impresa. Esso va inteso come un insieme, composto di strumenti e funzioni di ausilio al processo decisionale, quindi anche all'azione di governo dell'impresa, tra loro concatenati in una sequenza logico-operativa. Questo insieme di attività dovrà essere in grado di verificare il corretto funzionamento del sistema aziendale attraverso il monitoraggio di tutte le varie fasi, principalmente di quelle potenzialmente non idonee al perseguimento dell'efficacia e dell'efficienza. Il parametro con cui si misura l'efficienza è il valore economico, ovvero i costi sostenuti dall'impresa per l'ottenimento dei successivi ricavi. L'inefficienza dipende da anomalie funzionali che impediscono all'impresa di ottenere output superiori alle risorse impiegate. Due esempi di sistemi volti al monitoraggio delle attività aziendali sono i sistemi ERP (Enterprise resource planning) e i Data warehouse. L'ERP è un software di gestione che integra tutti i processi di business rilevanti di un'azienda (vendite, acquisti, gestione magazzino, contabilità ecc.) e nasce dall'esigenza delle aziende di affrontare un mercato globalizzato e di realizzare una forte integrazione tra tutte le applicazioni ed i dati, indipendentemente dalla loro posizione geografica o logica. I Data warehouse invece sono archivi informatici contenenti i dati di un'organizzazione, progettati per produrre facilmente analisi e relazioni utili ai fini decisionali-aziendali partendo da una mole di dati molto grande. Contestualmente alla nascita di mercati internazionali si presenta il problema di ottimizzare i tempi di produzione per essere più reattivi e competitivi sul mercato, infatti se fino a 50-60 anni fa le attività imprenditoriali operavano principalmente su scala regionale o al più nazionale, oggi grazie anche ad internet le aziende che offrono servizi aldilà dei confini sono molte. Ci si pone quindi il problema di dover soddisfare una domanda elevata in tempi brevi, e la gestione di flussi così elevati non è più sostenibile soltanto con regole di buon senso. Nasce così una collaborazione tra uomo e macchina, da una parte un calcolatore (oggi semplici pc) che elabora dati tramite algoritmi nati da nuove branche della matematica applicata e dall'altra parte un essere umano che analizza i risultati di tali calcoli e che è chiamato a prendere decisioni basandosi sugli stessi. Ma cosa si intende per processo ottimo? Un processo ottimo può essere definito come un insieme di attività che massimizza i profitti e/o minimizza i costi soddisfacendo contemporaneamente sia le aspettative del committente che tutti i vincoli tecnologici e organizzativi dell'azienda. Un uso corretto delle tecniche di ottimizzazione permette tutto questo garantendo una serie di vantaggi qualitativi e quantitativi che in generale si traducono in più elevati livelli di produttività e/o di servizio e in

un miglior uso delle risorse. L'applicazione riguarderà un'azienda telefonica ed in particolare il monitoraggio delle operazioni di quattro centrali sparse sul territorio torinese e l'ottimizzazione dei percorsi degli operatori al fine di concludere il maggior numero di operazioni durante una giornata lavorativa. Il monitoraggio farà riferimento alle tipologie di segnalazioni nel periodo selezionato, alle aree di riferimento delle quattro centrali e al resoconto sulle chiusure (operazioni portate a termine) e nuove segnalazioni su periodo mensile, trimestrale e quadrimestrale. Di seguito verrà fornita una spiegazione dettagliata sui vari fattori sotto controllo e le modalità di monitoraggio:

- **tipologie**: le tipologie fanno riferimento ai tipi di segnalazioni possibili sul territorio (Manutenzione, Generico Fonia, Danneggiamento, Router, Tensioni estranee in linea, Armadi e permutatori). Il monitoraggio viene fatto tramite un grafico a torta che indica la percentuale di nuove segnalazioni di ogni tipo nel periodo selezionato. Questo grafico permetterà di riscontrare anomalie qualora vi sia una prevalenza di segnalazioni riguardanti danneggiamenti o manutenzioni rispetto ad altre più comuni.

- **aree**: le aree fanno riferimento alle quattro centrali (TO-LINGOTTO, TO-LUCENTO, TO-CENTROURB, TO-CROCETTA). Anche in questo caso il monitoraggio viene fatto tramite un diagramma a torta, dove però ogni fetta indica la percentuale di nuove segnalazioni nella zona di competenza della relativa centrale nel periodo selezionato. Questo grafico permetterà di valutare quale zona è maggiormente interessata, ovvero quale area risulta essere la più critica in termini di nuove segnalazioni, permettendo di concentrare la maggior parte di operatori nella centrale di competenza.

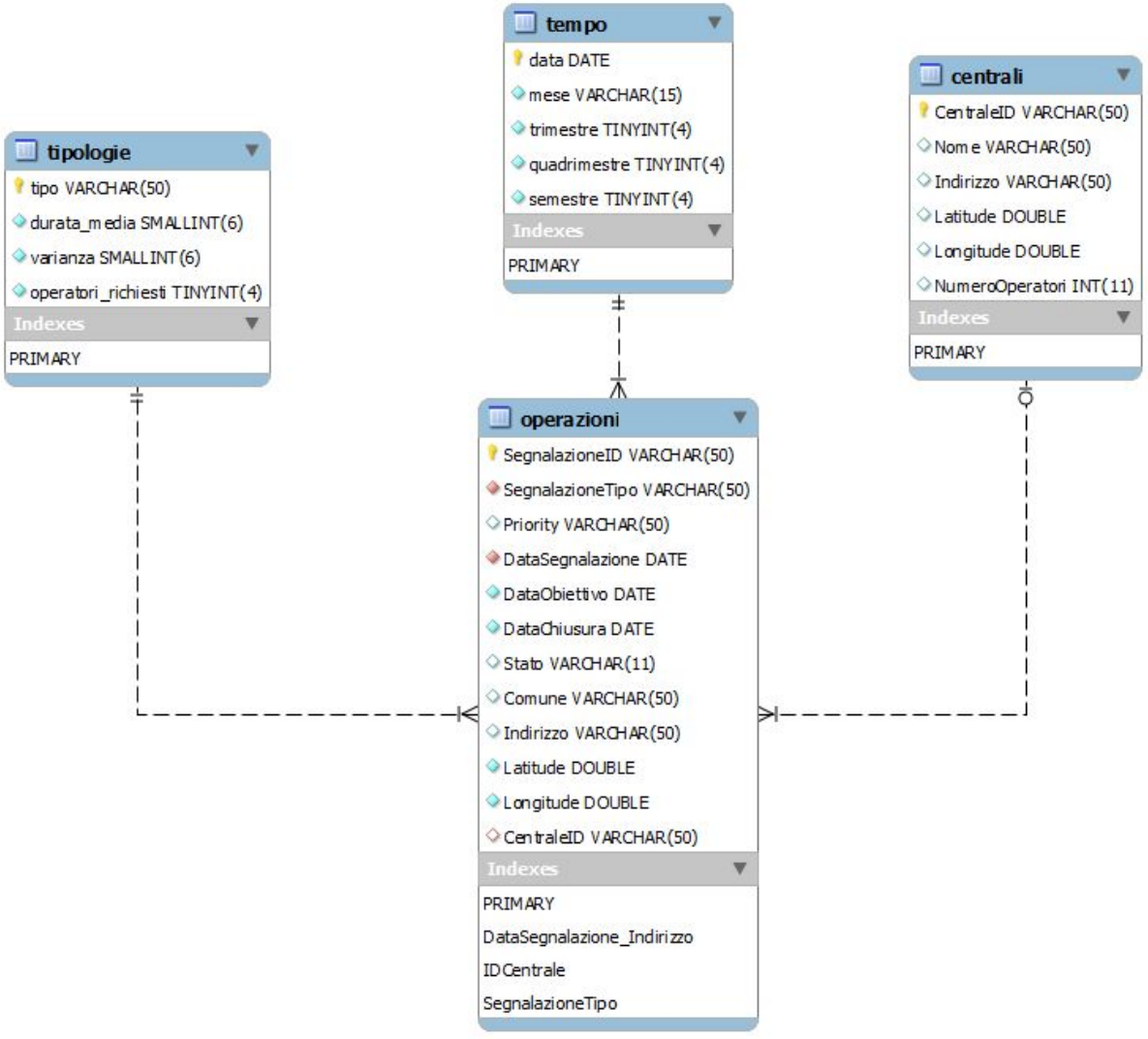
- **resoconto chiusure/nuove**: i resoconti fanno riferimento alle chiusure, termine tecnico per indicare le operazioni concluse entro la data obiettivo e le nuove segnalazioni nel periodo selezionato (Mese, Trimestre, Quadrimestre). Questo tipo di monitoraggio assume la forma di un'analisi fatta su un piccolo data warehouse estratto direttamente dallo stesso database e fornita all'utente sotto forma di due tabelle. Questo output permetterà di controllare l'andamento delle operazioni, individuando i periodi critici, ovvero quelli con maggior e minor carico. Permetterà inoltre di confrontare ogni singolo slot di tempo con la media del periodo selezionato.

Per quanto riguarda invece il problema di ottimizzazione, l'applicazione si propone di calcolare il percorso più rapido, che permetterà di concludere più operazioni possibili in giornata. Questo tipo di ottimizzazione è necessario per l'azienda che deve affrontare un numero elevato di domande giornaliere, senza un'adeguata programmazione degli spostamenti giornalieri infatti l'azienda accumulerebbe segnalazioni non risolte con conseguente malcontento degli utenti che ne risentirebbero in termini di qualità del servizio

per il quale pagano. Questo tipo di analisi verrà fornita come un output testuale riguardante gli spostamenti degli operatori durante la giornata con l'aggiunta della percentuale di operazioni svolte rispetto al totale da svolgere. Grazie a questo output si potranno riscontrare eventuali carenze di personale da risolvere ridistribuendo le risorse umane in maniera più conveniente sul territorio.

Descrizione del data-set

Diagramma del database:



Descrizione tabelle:

- **tipologie:** questa tabella rappresenta le tipologie di operazioni possibili nell'applicazione. Il campo *tipo* rappresenta la chiave primaria ed è rappresentato da una stringa che può assumere i valori Manutenzione, Generico Fonia, Danneggiamento, Router, Tensioni estranee in linea, Armadi e permutatori. *durata_media* e *varianza* rappresentano rispettivamente il tempo medio di completamento dell'operazione e la varianza, espresse entrambe in minuti. *operatori_richiesti* invece indica il numero di operatori richiesti per quell'operazione.

- **centrali:** tabella rappresentante le centrali disposte sul territorio. Il campo *CentraleID* rappresenta la chiave primaria della tabella ed è formato da una stringa rappresentante il codice della centrale, il campo *nome* che rappresenta il nome della centrale, il campo *indirizzo* indica l'ubicazione della centrale, mentre i campi *latitude* e *longitude* indicano le coordinate geografiche. Infine NumeroOperatori indica il numero di operatori a disposizione della centrale.

- **tempo**: tabella che serve per distinguere i vari periodi dell'anno. Il campo *data* è la chiave primaria ed è rappresentato da un attributo di tipo DATA. Il campo *mese* indica il mese di riferimento della data, mentre i campi *trimestre*, *quadrimestre* e *semestre* sono rappresentati da un numero che indica di quale trimestre, quadrimestre, semestre si tratta.

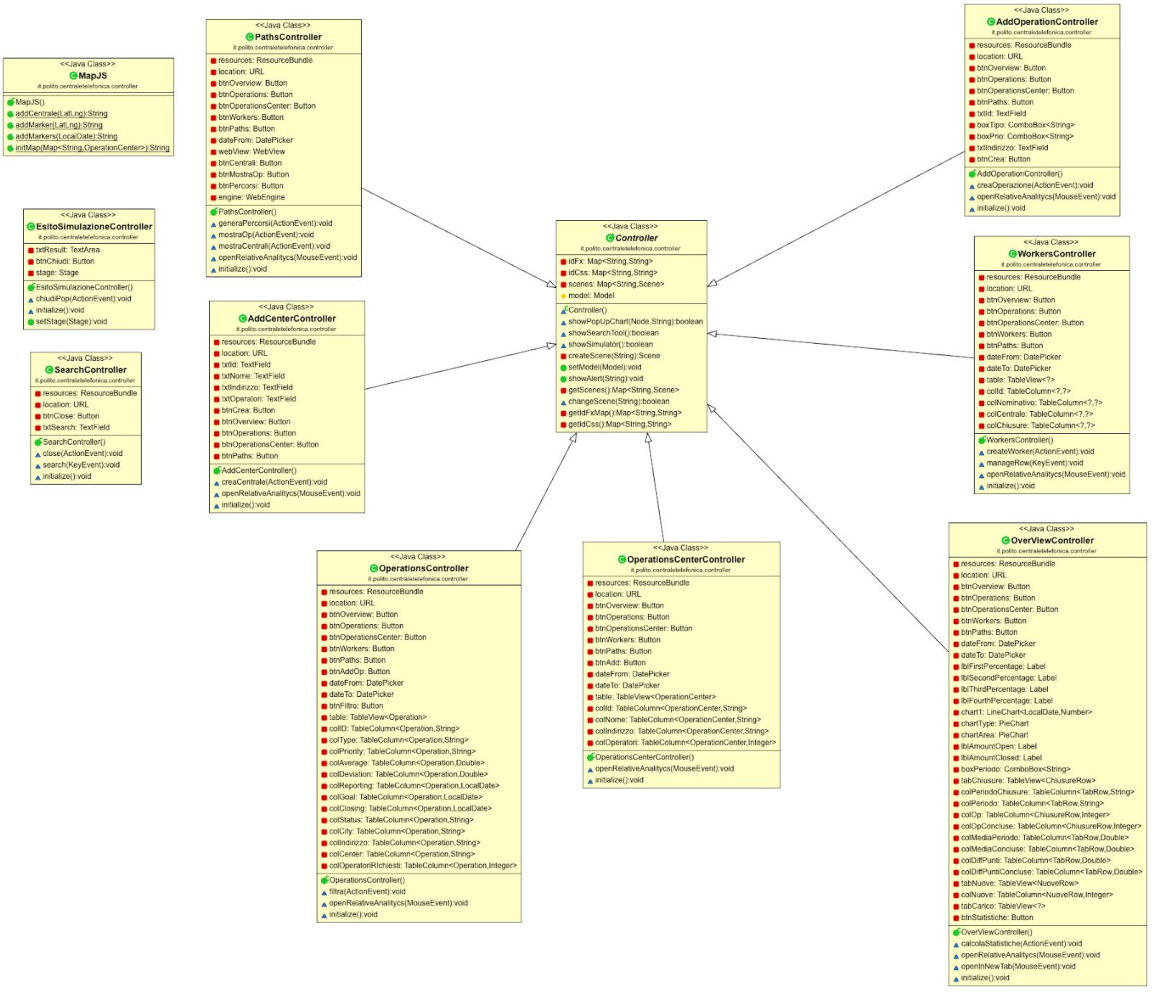
- **operazioni:** tabella rappresentante le operazioni da portare a termine. In questa tabella troviamo in chiave primaria *SegnalazioneID* che indica appunto l'identificativo dell'operazione tramite una stringa. Come chiavi esterne invece troviamo *SegnalazioneTipo* che fa riferimento alla tabella tipologie e *CentraleID* che fa riferimento alla tabella centrale. Il campo *Priority* indica la priorità dell'operazione, *DataSegnalazione* fa riferimento alla data nella quale è stata effettuata la segnalazione da un utente o da un operatore, mentre *DataObiettivo* fa riferimento alla data entro la quale ci si aspetta di terminare l'operazione. Contestualmente *DataChiusura* indica la data di effettiva conclusione dell'operazione ed il campo *Stato* indica se l'operazione è stata conclusa o meno. Infine i campi *Comune*, *Indirizzo*, *Latitude* e *Longitude* indicano la posizione dell'operazione.

Descrizione strutture dati, algoritmi e diagrammi classi

L’applicazione è stata scritta facendo uso dei seguenti linguaggi: Java, JavaFX, HTML, CSS e JavaScript ed implementa due tra i pattern applicativi più comuni, il pattern MVC (Model-View-Controller) ed il pattern DAO (Data Access Object). Le risorse applicative sono divise in quattro package: **it.polito.centraletelefonica.controller** che contiene tutti i controller delle relative views, **it.polito.centraletelefonica.db** che contiene la logica applicativa riguardante la comunicazione con il database, **it.polito.centraletelefonica.main** contenente tutti i file .fxml con i quali è stata strutturata la grafica e **it.polito.centraletelefonica.model** che contiene la logica applicativa.

it.polito.centraletelefonica.controller

Schema UML del package, nel progetto verrà allegato un’immagine .png per visualizzare meglio lo schema



come si può notare tutti i controller ereditano dalla classe Controller che contiene metodi utili a tutte le altre classi, per esempio il metodo changeScene riceve come input una stringa rappresentante l'id del nodo (semplice button) che apre la relativa scena :

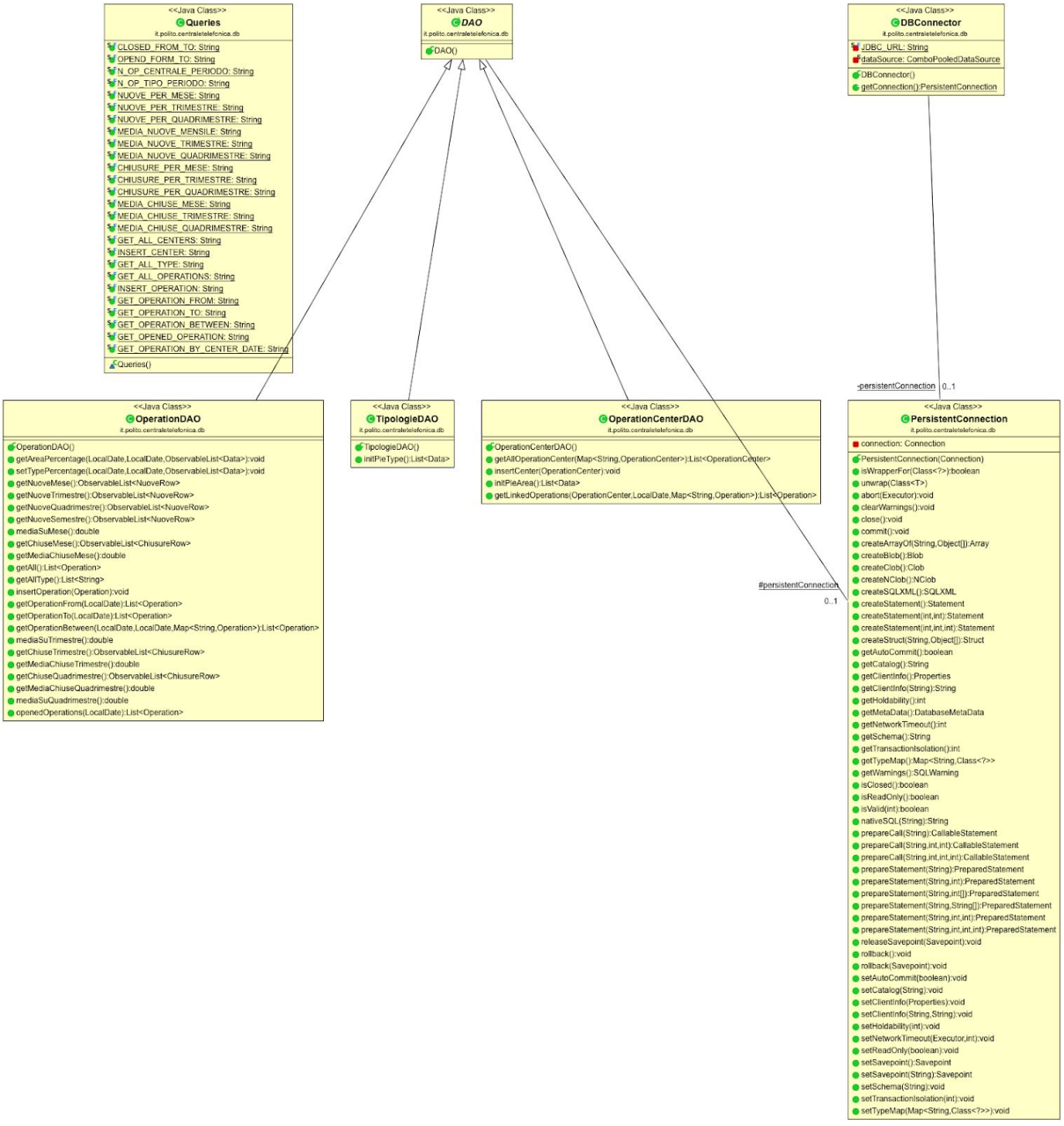
```
boolean changeScene(String nodeID) {
       Scene scene = createScene(nodeID);
       if (scene != null) {
           App.getStage().setScene(scene);
           return true;
               }
       return false;
}
```

```
       private Scene createScene(String nodeID) {
          if (!getScenes().containsKey(nodeID)) {
            if (getIdFxMap().containsKey(nodeID)) {
               String fileName = getIdFxMap().get(nodeID);
               FXMLLoader loader = new  FXMLLoader(App.class.getResource(fileName));
               BorderPane root;
               Scene scene = null;
               try {
                    root = loader.load();
                    Controller controller = loader.getController();
                    if (controller != null) {
                        controller.setModel(new Model());
                        }
                   double previousWidth = App.getStage().getScene().getWidth();
                   double previousHeight = App.getStage().getScene().getHeight();
                   scene = new Scene(root, previousWidth, previousHeight);
                   if (getIdCss().containsKey(nodeID)) {
                       String cssFileName = getIdCss().get(nodeID);
                       scene.getStylesheets()
                       .add(App.class.getResource(cssFileName)
                       .toExternalForm());
                       getScenes().put(nodeID, scene);
                       return scene;
                    }
                  } catch (IOException e) {
                       e.printStackTrace();
                    }
               }

               }

               return getScenes().get(nodeID);

       }
```

ogni controller come da specifica nel pattern MVC fa da ponte tra la View e la logica.

it.polito.centraletelefonica.db

Schema UML del package, nel progetto verrà allegato un’immagine .png per visualizzare meglio lo schema



In questo package sono presenti le classi che hanno il compito di interagire con il database. Innanzitutto troviamo la classe DBConnector che offre una connessione persistente al database tramite il pattern SingleTone

```
public class DBConnector {

  private static final String JDBC_URL =
"jdbc:mysql://localhost/centraletelefonica?user=root&zeroDateTimeBehavior=convertToNull";
  private static ComboPooledDataSource dataSource;
  private static PersistentConnection persistentConnection;

        public static PersistentConnection getConnection() throws SQLException {

           if (persistentConnection == null) {
               dataSource = new ComboPooledDataSource();
               dataSource.setJdbcUrl(JDBC_URL);
               persistentConnection = new PersistentConnection(dataSource.getConnection());
               return persistentConnection;
           }

          return persistentConnection;

         }

}
```

Nel package sono presenti tante classi DAO quanti sono i java bean da rappresentare. Ognuna di queste classi ha il compito di fare da ponte tra la logica ed il database eseguendo le query fornite dalla classe Queries, che è una classe contenente soltanto stringhe costanti che definiscono le query. Di seguito verranno riportati alcuni esempi:

```
public static final String MEDIA_CHIUSE_MESE = "select sum(sub1.op_mese)/12 avg_mensile \r\n"
       + "from(\r\n"
       + "select count(t.mese) op_mese \r\n" + "from operazioni, tempo t\r\n"
       + "where DataChiusura <= DataObiettivo \r\n" + "and DataChiusura <> \"0000-00-00\"\r\n"
       + "and Stato = \"Closed\"\r\n" + "and DataSegnalazione = t.`data`\r\n"
       + "group by t.mese\r\n"
       + "order by t.mese) sub1;";

public static final String MEDIA_CHIUSE_TRIMESTRE = "select sum(sub1.op_trimestre)/4
avg_trimestrale\r\n"
       + "from(\r\n" + "select count(t.trimestre) op_trimestre from operazioni, tempo t\r\n"
       + "where DataChiusura <= DataObiettivo\r\n" + "and DataChiusura <> \"0000-00-00\"\r\n"
       + "and Stato = \"Closed\"\r\n" + "and DataSegnalazione = t.`data` \r\n" + "group by t.mese
\r\n"
       + "order by t.mese) sub1;";
```

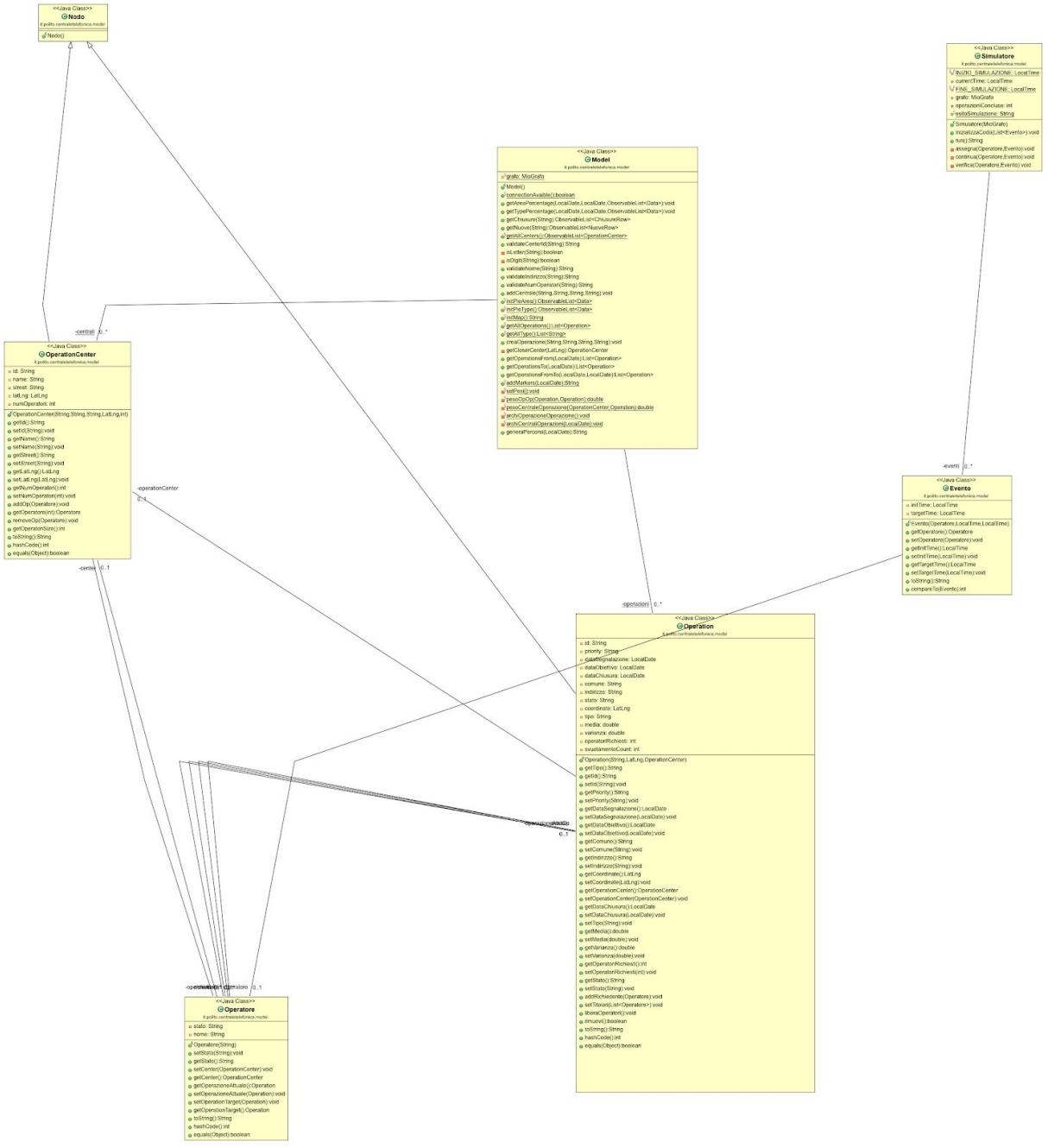
it.polito.centraletelefonica.main

Nel package main si trova la classe App ed i file .fxml che rappresentano le view delle varie scene. La classe App inoltre contiene il metodo main che lancia l'applicazione

```
public class App extends Application {

        private static Stage stage;

        @Override
        public void start(Stage stage) {

        App.stage = stage;

           try {

                FXMLLoader loader = new FXMLLoader(getClass().getResource("OverView.fxml"));
                BorderPane root = loader.load();
                Scene scene = new Scene(root, 1200, 600);
               scene.getStylesheets().add(getClass().
               getResource("application.css").toExternalForm());
                App.stage.setScene(scene);
                App.stage.centerOnScreen();
                App.stage.setTitle("Centrale Telefonica");
                OverViewController controller = loader.getController();
                Model model = new Model();
                controller.setModel(model);
                App.stage.show();

                }

                catch (Exception e) {
                   e.printStackTrace();
                }
        }

        public static void main(String[] args) {
                launch(args);
        }

        public static Stage getStage() {
                return stage;
        }
}
```

it.polito.centraletelefonica.model

Nel package model è contenuta la logica applicativa che risiede nella classe Model e dei vari Java Bean. Di seguito verrà fornito il diagramma UML delle principali classi che compongono tale package (nel progetto verrà fornito sotto forma di immagine .png)



I Java Bean sono rappresentati dalle classi Operation, OperationCenter ed Operatore. Operation rappresenta le operazioni ed è la rappresentazione in linguaggio a oggetti della relazione operazioni-tipologie del database. In maniera analoga OperationCenter rappresenta le centrali, mentre la classe Operatore rappresenta in maniera fittizia un operatore in quanto tale oggetto è caratterizzato semplicemente da un id. La struttura dati utilizzata per contenere le centrali e le operazioni è un grafo rappresentato dalla classe MioGrafo. Il grafo in questione è non orientato in quanto non avrebbe senso definire un verso al collegamento tra le operazioni connesse tra di loro non conoscendo a priori il percorso degli operatori. Gli archi del grafo sono pesati ed il peso è dato da tre componenti: il tempo di percorrenza tra un'operazione e l'altra, il tempo medio necessario allo svolgimento dell'operazione ed infine il numero di operatori richiesti. Nello specifico il peso = (tempo di percorrenza + tempo svolgimento operazione) * numero di operatori richiesti. Si è deciso moltiplicare per il numero di operatori richiesti in quanto si vuole rendere più svantaggiose operazioni che tengono occupati molti operatori che quindi non possono muoversi sul territorio. Tutte le operazioni sono connesse tra di loro. Di particolare interesse è il metodo generaPercorsi della classe Model che serve a calcolare e simulare il percorso degli operatori in una determinata giornata. Presento quindi il codice relativo a tali passaggi che verrà di seguito commentato:

```
public String generaPercorsi(LocalDate localDate) {

        // rigenera il grafo
           grafo = new MioGrafo();
           OperationDAO dao = new OperationDAO();
           OperationCenterDAO centerDao = new OperationCenterDAO();
           List<Operation> operazioniLista = new LinkedList<>(
           dao.getOperationBetween(value, value.minusDays(5), operazioni));
           Graphs.addAllVertices(grafo, operazioniLista);
           Graphs.addAllVertices(grafo, centerDao.getAllOperationCenter(centrali));
           archiCentraliOperazioni(value);
           archiOperazioneOperazione();
           setPesi();


      // Creo gli operatori e li assegno alle centrali

      int opId = 0;

      for (Iterator<OperationCenter> iterator = centrali.values().iterator(); iterator.hasNext();) {
           OperationCenter center = (OperationCenter) iterator.next();
           for (int i = 0; i < center.getNumOperatori(); i++) {
                Operatore operatore = new Operatore("Operatore " + opId++);
                center.addOp(operatore);
                operatore.setCenter(center);
                operatore.setStato("libero");
                }
                }

                // Inizialmente gli operatori partono dalle centrali e si dirigono verso le
                // operazioni piu' vicine(peso minimo) fino a svuotare le centrali(se
                // necessario)

                Simulatore simulatore = new Simulatore(grafo);
                List<Evento> eventi = new LinkedList<>();
```

```
      for (Iterator<OperationCenter> iterator = centrali.values().iterator(); iterator.hasNext();) {
                OperationCenter center = (OperationCenter) iterator.next();
                ClosestFirstIterator<Nodo, DefaultWeightedEdge> closest = new
ClosestFirstIterator<>(grafo, center);
                // il primo lo mando a vuoto perché corrisponde al nodo di partenza ovvero la
                // centrale stessa
                closest.next();
                // ciclo fino a svuotare la centrale e/o ad esaurire le operazioni giornaliere
                int assegnate = 0;
                int daAssegnare = Graphs.neighborListOf(grafo, center).size();
                while (center.getOperatoriSize() > 0 && assegnate < daAssegnare) {
                       Operation nextOperation = (Operation) closest.next();
                       int opRichiesti = nextOperation.getOperatoriRichiesti();
                       for (int i = 0; i < opRichiesti; i++) {
                            Operatore operatore = center.getOperatore(i);
                            center.removeOp(operatore);
                            DefaultWeightedEdge edge = grafo.getEdge(center, nextOperation);
                        // gli operatori partono verso le centrali quindi l'inizio dell'evento
                        // corrisponde alle 8:00 del mattino, mentre il target corrisponde
                       //all'orario d'arrivo sul posto

                        double tempo = (grafo.getEdgeWeight(edge) /
                       nextOperation.getOperatoriRichiesti()) - nextOperation.getMedia() * 60;

                        Evento evento = new Evento(operatore, LocalTime.of(8, 00),
                                                   LocalTime.of(8, 00).plusSeconds((long) tempo));
       // una volta mossi verso l'operazione elimino l'arco per evitare di ritornare
       // sulla stessa operazione, cosa che non avrebbe senso
          grafo.removeEdge(edge);
       // delega: l'operatore deve sapere verso quale operazione muoversi e
       // l'operazione deve conoscere l'operatore richiedente.
          operatore.setOperationTarget(nextOperation);
          operatore.setStato("in viaggio");
          eventi.add(evento);

                              }

                        }

                }

         simulatore.inizializzaCoda(eventi);
         String esito = simulatore.run();
         return esito;

       }
```

```
        /**
         * si suppone che alle 8, inizio della giornata gli operatori si trovino sul
         * posto della prima operazione da svolgere in giornata
         */

       public String run() {

          int opIniziali = grafo.vertexSet().size();

          // clausola d'uscita: fine giornata.
          while (currentTime.compareTo(FINE_SIMULAZIONE) < 0) {
```

```
                // scagliono il tempo in secondi
                currentTime = currentTime.plusSeconds(1);

                Evento ev = eventi.poll();
                Operatore op = ev.getOperatore();

               if (ev.getTargetTime().compareTo(FINE_SIMULAZIONE) <= 0
                                       && ev.getTargetTime().compareTo(INIZIO_SIMULAZIONE) > 0) {

                  switch (op.getStato()) {
                  // se un operatore è occupato verifico se ha completato l'operazione
                        case "occupato":
                             verifica(op, ev);
                             break;

                        case "in viaggio":
                   // se un operatore è in viaggio devo controllare che sia ancora necessario che
                   // si muova verso la destinazione, se non più necessario devo fargli cambiare
                   // rotta
                         continua(op, ev);
                         break;

                         case "libero":
               // assegno all'operatore una nuova destinazione
                         assegna(op, ev);

                                 default:
                                         break;
                                 }

                         }

               }

               // alla fine stampo statistiche
               double percentualeConclusa = (double) (operazioniConcluse / (double) opIniziali);
               esitoSimulazione += "\n Percentuale operazioni portate a termine: "
                                 + new DecimalFormat(".##").format(percentualeConclusa) + " %";
               return esitoSimulazione;

        }

        private void assegna(Operatore op, Evento ev) {

               ClosestFirstIterator<Nodo, DefaultWeightedEdge> closest = new
ClosestFirstIterator<Nodo, DefaultWeightedEdge>(
               grafo, op.getOperazioneAttuale());

               // quando tutti gli operatori lasciano il luogo dell'operazione elimino il nodo
               // dal grafo
               if (op.getOperazioneAttuale().rimuovi())
                   grafo.removeEdge(grafo.getEdge(op.getOperazioneAttuale(),
op.getOperationTarget()));

               // il primo giro a vuoto perché corrisponde alla partenza

               if (closest.hasNext()) {

                         closest.next();
                         Operation nextOp = (Operation) closest.next();
```

```
                        double distance = LatLngTool.distance(
                                        new
com.javadocmd.simplelatlng.LatLng(op.getOperazioneAttuale().getCoordinate().lat,

op.getOperazioneAttuale().getCoordinate().lng),
                                        new
com.javadocmd.simplelatlng.LatLng(op.getOperationTarget().getCoordinate().lat,

op.getOperationTarget().getCoordinate().lng),
                                        LengthUnit.METER);

                        // Supponiamo una velocita' di 14 m/s, un po' piu' di 50 km/h.
                        double secondi = distance / 14;
                        op.setStato("in viaggio");
                        op.setOperationTarget(nextOp);
                        Evento evento = new Evento(op, currentTime, currentTime.plusSeconds((long)
secondi));
                                esitoSimulazione += evento.toString() + "\n";
                                eventi.add(evento);
                        }

                }


        private void continua(Operatore op, Evento ev) {

                // se sul posto ci sono già gli operatori richiesti l'operatore deve cambiare
                // rotta

                Operation nextOp = op.getOperationTarget();

                if (nextOp.getStato() == "IN_CORSO") {
                        ClosestFirstIterator<Nodo, DefaultWeightedEdge> closest = new
ClosestFirstIterator<>(grafo, nextOp);
                        // salto il primo nodo perché corrisponde alla partenza
                        closest.next();
                        Operation nextDestination = (Operation) closest.next();
                        // nonostante non sia effettivamente arrivato sul posto suppongo che invece
lo
                        // sia per facilitare i calcoli del percorso da un punto ad un altro non
potendo
                        // conoscere la posizione esatta di un punto intermedio tra un'operazione
ed
                        // un'altra
                        DefaultWeightedEdge edge = grafo.getEdge(nextOp, nextDestination);
                        op.setOperationTarget(nextDestination);
                        double tempo = (grafo.getEdgeWeight(edge) /
nextDestination.getOperatoriRichiesti())
                                        - nextDestination.getMedia() * 60;
                        op.setStato("in viaggio");
                        Evento evento = new Evento(op, currentTime, currentTime.plusSeconds((long)
tempo));
                                eventi.add(evento);
                }

                // se invece l'operazione non ha raggiunto il numero di operatori e l'operatore
                // arriva sul posto allora l'operazione assume uno stato in corso e l'operatore
                // diventa occupato

                if (nextOp.getStato() != "IN_CORSO") {
```

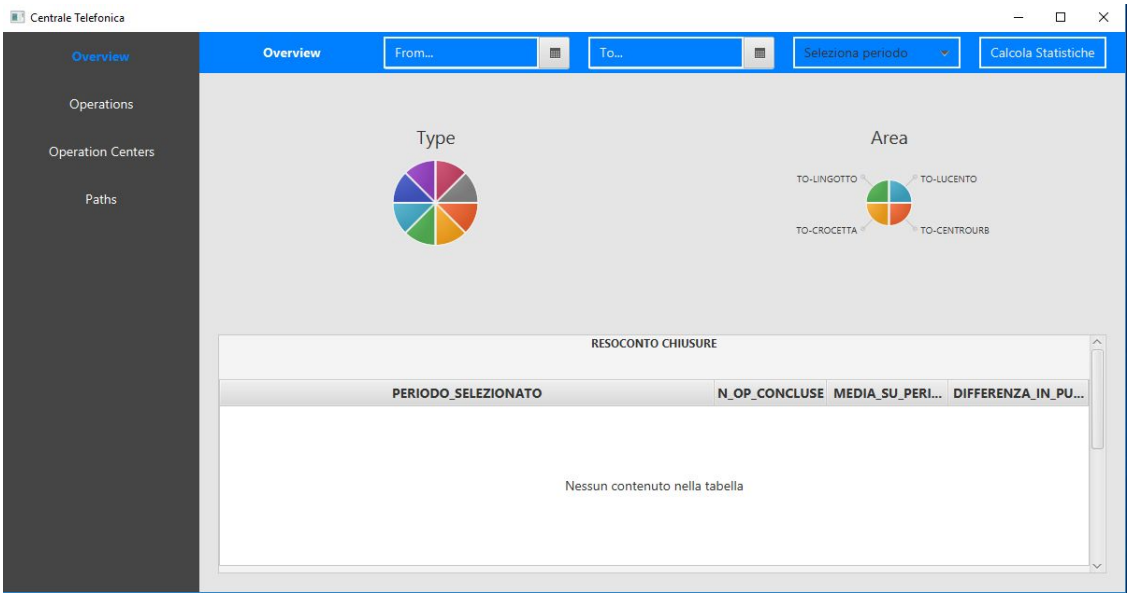
```
                        // se l'operatore arriva sul posto si occupa dell'operazione
                        if (currentTime.compareTo(ev.getTargetTime()) >= 0 && op.getStato() !=
"occupato") {
                                nextOp.addRichiedente(op);
                                // tempo per il quale l'operatore sarà occupato
                                double secRichiesti = op.getOperationTarget().getMedia() * 60;
                                op.setStato("occupato");
                                op.setOperazioneAttuale(nextOp);
                                Evento eve = new Evento(op, currentTime,
currentTime.plusSeconds((long) secRichiesti));
                                esitoSimulazione += eve.toString() + "\n";
                                eventi.add(eve);
                        }
                        // altrimenti re-inserisco l'operazione in coda perché non trattata
                        else {
                                Evento evento = new Evento(op, currentTime, ev.getTargetTime());
                                eventi.add(evento);
                        }

                }

        }

        private void verifica(Operatore op, Evento ev) {

                // operazione terminata
                if (currentTime.compareTo(ev.getTargetTime()) >= 0) {
                        // gli operatori diventano liberi
                        op.getOperationTarget().liberaOperatori();
                        op.setStato("libero");
                        Evento evento = new Evento(op, currentTime, currentTime);
                        esitoSimulazione += evento.toString() + "\n";
                        operazioniConcluse++;
                        eventi.add(evento);
                }
                // altrimenti aggiungo di nuovo l'evento non trattato
                else {
                        Evento evento = new Evento(op, currentTime, ev.getTargetTime());
                                eventi.add(evento);
                }

        }
```

Nella prima fase del metodo generaPercorsi vengono creati gli operatori, tanti quanto ce ne sono nelle centrali e tale numero viene prelevato dalla tabella *centrali* del database. Ogni operatore viene assegnato ad una centrale ed assume lo stato di libero, che sta a significare che l'operatore non sta svolgendo nessuna attività. A questo punto vi è un doppio ciclo, uno sulle centrali e uno sul numero di operazioni che sono assegnate a quella centrale. Così facendo si prelevano da ogni centrale ottenuta tramite il primo ciclo gli operatori che verranno inviati sul territorio. Il ciclo interno itera fin quando tutti gli operatori sono stati mossi sul territorio e/o non ci sono più operazioni da svolgere. Si creano quindi degli eventi caratterizzati da un operatore, un orario inizio ed un orario finale. Inizialmente tutti gli eventi avranno come tempo di inizio le 8:00 che corrisponde all'inizio della giornata lavorativa ed avranno come orario finale l'orario di inizio giornata maggiorato di un quantitativo di secondi

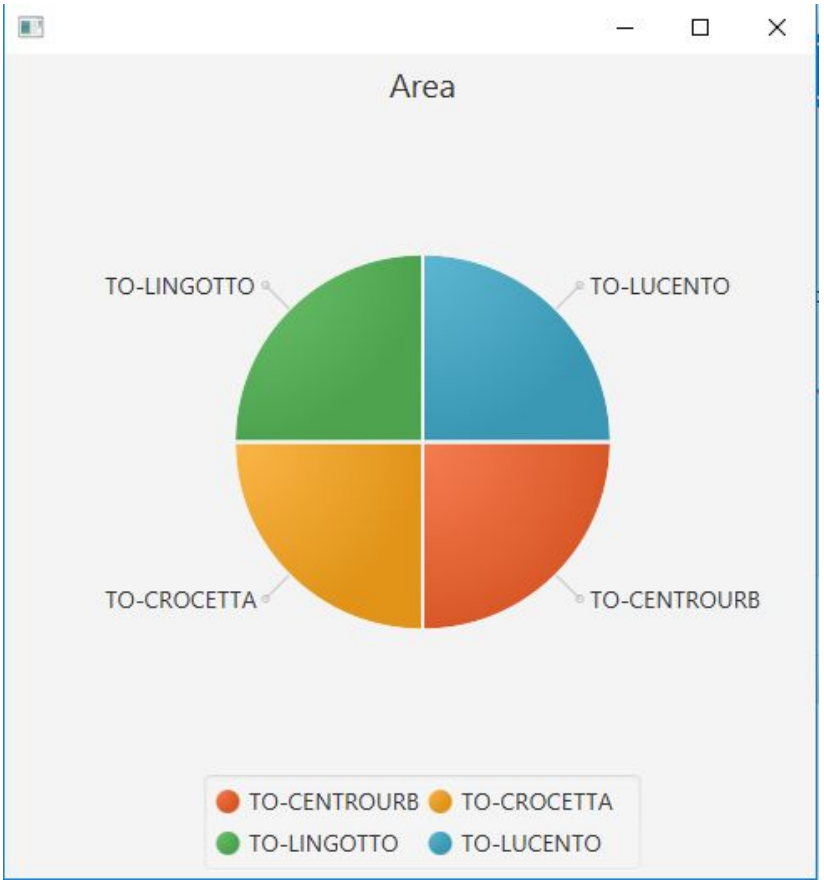
pari ai secondi necessari all'operatore per raggiungere la prima operazione dalla centrale. L'operazione di destinazione verrà scelta tramite il metodo closest della classe ClosestFirstIterator<V,E>, di seguito verrà eliminato l'arco tra centrale e l'operazione, in quanto non avrebbe senso tornare in centrale, e l'operatore assume lo stato "in viaggio". Tutti gli eventi vengono aggiunti ad una lista che verrà fornita al simulatore precedentemente creato. Infine viene chiamato il metodo run() che da l'inizio alla simulazione. Tramite il metodo run() della classe Simulatore possiamo simulare l'andamento della giornata lavorativa prelevando gli eventi dalla Queue nella quale erano stati inseriti e scandendo il tempo in secondi. Tale simulazione ha come inizio l'orario di inizio giornata e come fine l'orario lavorativo oltre il quale non si svolgeranno più operazioni. Troviamo uno switch sullo stato dell'operatore, infatti ogni operatore può essere libero, se non sta svolgendo nessuna attività, in viaggio o occupato se sta svolgendo una qualche operazione sul territorio. Se l'operatore risulta occupato viene chiamato il metodo verifica che ha il compito di verificare se l'operazione è conclusa o meno. Se l'operazione è conclusa tutti gli operatori occupati su quell'operazione assumono lo stato di "libero" potendo così essere assegnati ad altre operazioni creando un nuovo evento che ha come inizio e fine il current time in quanto si vuole assegnare l'operatore il prima possibile , se così non fosse gli operatori rimangono occupati e si crea un evento che ha come inizio il current time, mentre ha come fine il precedente orario definito come target. Se l'operatore risulta essere in viaggio viene chiamato il metodo continua che si occupa di verificare se l'operatore deve continuare a dirigersi verso l'operazione di destinazione o deve cambiare rotta nel caso in cui un operatore fosse già arrivato sul posto. Infine se un operatore risulta essere libero allora gli si assegna un'operazione, se presente, il suo stato diventa "in viaggio" e si crea un evento da aggiungere alla coda. Al termine della simulazione viene stampato un elenco dei movimenti degli operatori sul territorio e la percentuale di operazioni svolte rispetto a quello da svolgere.

Alcune videate dell’applicazione

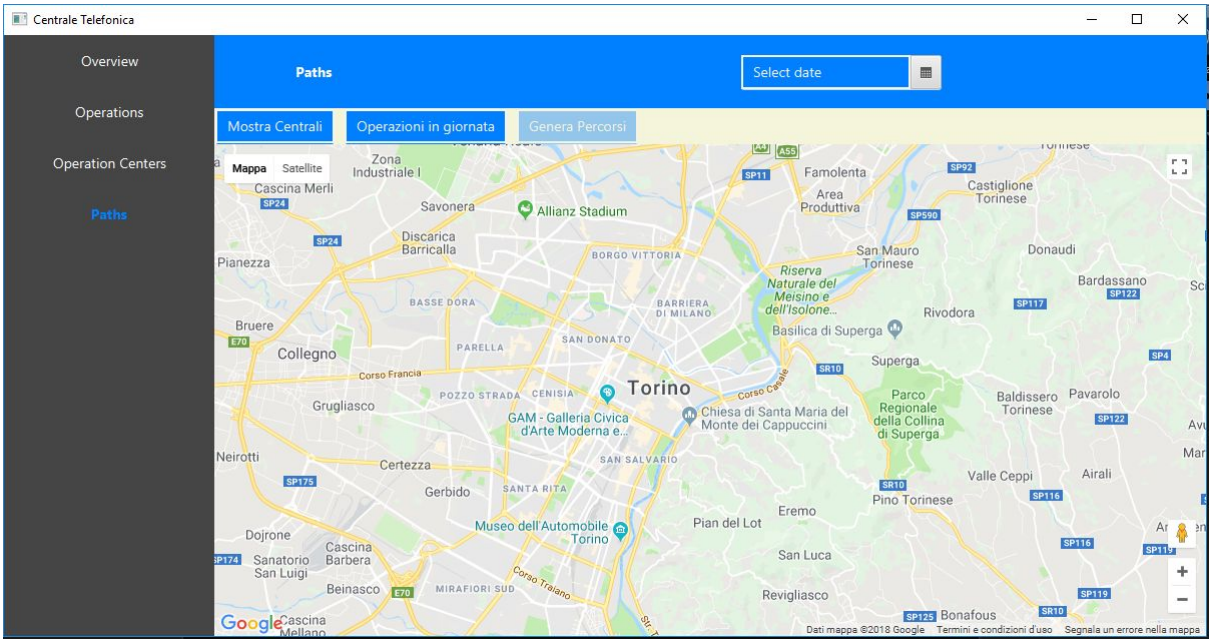
Scena iniziale: Overview



Pie chart “Area”



Paths



link al video: [CentraleTelefonica]

Tabella con risultati ottenuti

Data Simulazione	Operatori TO-CENTROURB	Operatori TO-CROCETTA	Operatori TO-LINGOTTO	Operatori TO-LUCENTO	% Operazioni concluse	Tempo di esecuzione [s]
13-11-2017	50	50	50	50	100%	5.748317518
13-11-2017	45	45	45	45	100%	6.288954258
13-11-2017	40	40	40	40	100%	6.517298903
13-11-2017	35	35	35	35	100%	6.951032068
13-11-2017	30	30	30	30	100%	7.152987532
13-11-2017	25	25	25	25	93.53%	7.70544714
14-11-2017	50	50	50	50	100%	12.978937446
14-11-2017	45	45	45	45	100%	12.834108406
14-11-2017	40	40	40	40	100%	12.990294915
14-11-2017	35	35	35	35	95.16%	12.074940266
14-11-2017	30	30	30	30	84.81%	13.162814989
14-11-2017	25	25	25	25	69.45%	13.373802111

Valutazioni sui risultati ottenuti e conclusioni

Le prove effettuate riportate in tabella hanno come prima data dei test il 13-11-2017 perché ci poniamo nel caso in cui questa sia la data odierna, inoltre si suppone che tutte le operazioni precedenti a quella data siano state concluse. Nelle condizioni iniziali , ovvero quando ogni centrale possiede 50 operatori, l'algoritmo ci permette di concludere la totalità di nuove segnalazioni entro metà giornata. Anche il tempo di esecuzione di 5 secondi è confortevole e sembra abbastanza stabile al variare del numero di operatori tra le centrali, questo dimostra che l'algoritmo è poco correlato al numero di operatori presenti. Man mano che diminuiscono gli operatori il tempo di completamento delle operazioni aumenta fino a far scendere la percentuale di operazioni concluse sotto il 100%. L'algoritmo segnala come soglia 30 operatori/centrale sotto la quale si rischia di non portare a termine le operazioni in giornata. In secondo luogo si effettua una prova al 14-11-2017 per simulare il caso in cui si abbia un carico lavorativo arretrato pari al carico di una giornata (quello del 13-11-2017). In questo caso si può notare che la soglia di operatori/centrali sale da 25 a 40 indicandoci che forse la soglia di 30 potrebbe essere rischiosa. Per quanto riguarda il tempo di esecuzione dell'algoritmo invece nella seconda prova notiamo che ora è quasi raddoppiato, sintomo che esiste una forte correlazione tra il numero di operazioni, quindi vertici nel grafo, e il tempo di calcolo. Date le semplificazioni dovute effettuare il numero iniziale di 50 operatori per centrale sembrerebbe essere dunque adeguato. Le criticità generate dalle semplificazioni che inficiano la bontà dei risultati ottenuti sono le seguenti:

- La distanza tra le operazioni viene calcolata in linea d'aria, mentre sarebbe necessario accedere ad informazioni più dettagliate sul percorso da effettuare tra un punto ed un altro
- La velocità risulta essere costante ed impostata ad un valore limite di 50 km/h. Non si hanno dati sull'andamento degli operatori e quindi si è impostata la soglia stradale massima, inoltre non si conoscono dati sul traffico e quindi si suppone che un operatore si muova alle otto del mattino come si muove in orari meno trafficati.
- Si è supposto che un'operatore non faccia pause, nel caso le facesse, come da diritto, si perderebbero 1-2 ore di lavoro al giorno.
- Si è supposto che il cliente sia disponibile a qualsiasi orario del giorno, non considerando che spesso gli interventi negli edifici si effettuano concordando una data e un orario con il cliente, questo fa ritardare sicuramente la data di completamento dell'operazione.
- Si è considerato che ogni operatore sappia portare a termine qualsiasi tipo di operazione.

Quest’opera è distribuita con licenza Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo 4.0 Internazionale.
Copia della licenza consultabile al sito web: Copia della licenza