Università degli Studi di Napoli
"Parthenope"
Dipartimento di Scienze e Tecnologie
Corso di Laurea in Informatica

Corso di Programmazione III

PrevenTech

Raffaele Attanasio 0124001695
Dario Musella 0124001799
Raffaele Venuso 0124001754

Docenti: prof. Angelo Ciaramella, prof. Raffaele Montella
Anno Accademico 2019/2020

Indice

Capitolo 1

Descrizione del progetto

1.1 Presentazione dell'idea

La seguente relazione illustra l'implementazione del progetto proposto. Si è voluto proporre l'estensione di un progetto "fulcro" già discusso in una precedente sede d'esame, dove il progetto *prevenTech* ha voluto porre l'accento su una questione di rilevanza sociale. Il progetto nato come sito web consente di ricercare gli erogatori di profilattici con le annesse farmacie più vicini alla propria zona.

Per il sito web è stato utilizzato il componente *Leaflet* per l'implementazione delle mappe, mentre i segnaposti corrispondenti ai distributori e alle farmacie sono gestiti da un database non relazionale. Questa estensione di progetto propone un applicativo back-office atto ad offrire un'interfaccia che automatizzi le operazioni gestionali che muovono il database MongoDB. In questo elaborato si farà riferimento ai *distributori* e alle *farmacie* in maniera del tutto equivalente, in quanto lo sviluppo di questo progetto si è incentrato prettamente sulle funzionalità gestionali.

Le operazioni utente che sono state implementate sono:

- **Inserimento** di una nuova tupla nel database
- **Cancellazione** di una tupla dal database
- **Annullamento** delle azioni effettuate

1.2 Requisiti del progetto

Il progetto è stato implementato attenendosi alle linee guida richieste dalla consegna. Il codice è stato stilato basando il funzionamento dello stesso su 7 design pattern: *Builder*, *Command*, *Facade*, *Iterator*, *Memento*, *Singleton* e *State*. La base di dati utilizzata per il progetto è la medesima del progetto di riferimento, la quale contiene già tutti i dati campione e gli attributi necessari per il funzionamento della mappa.

Il codice, opportunamente commentato prevede l'implementazione di una interfaccia grafica JavaFX.

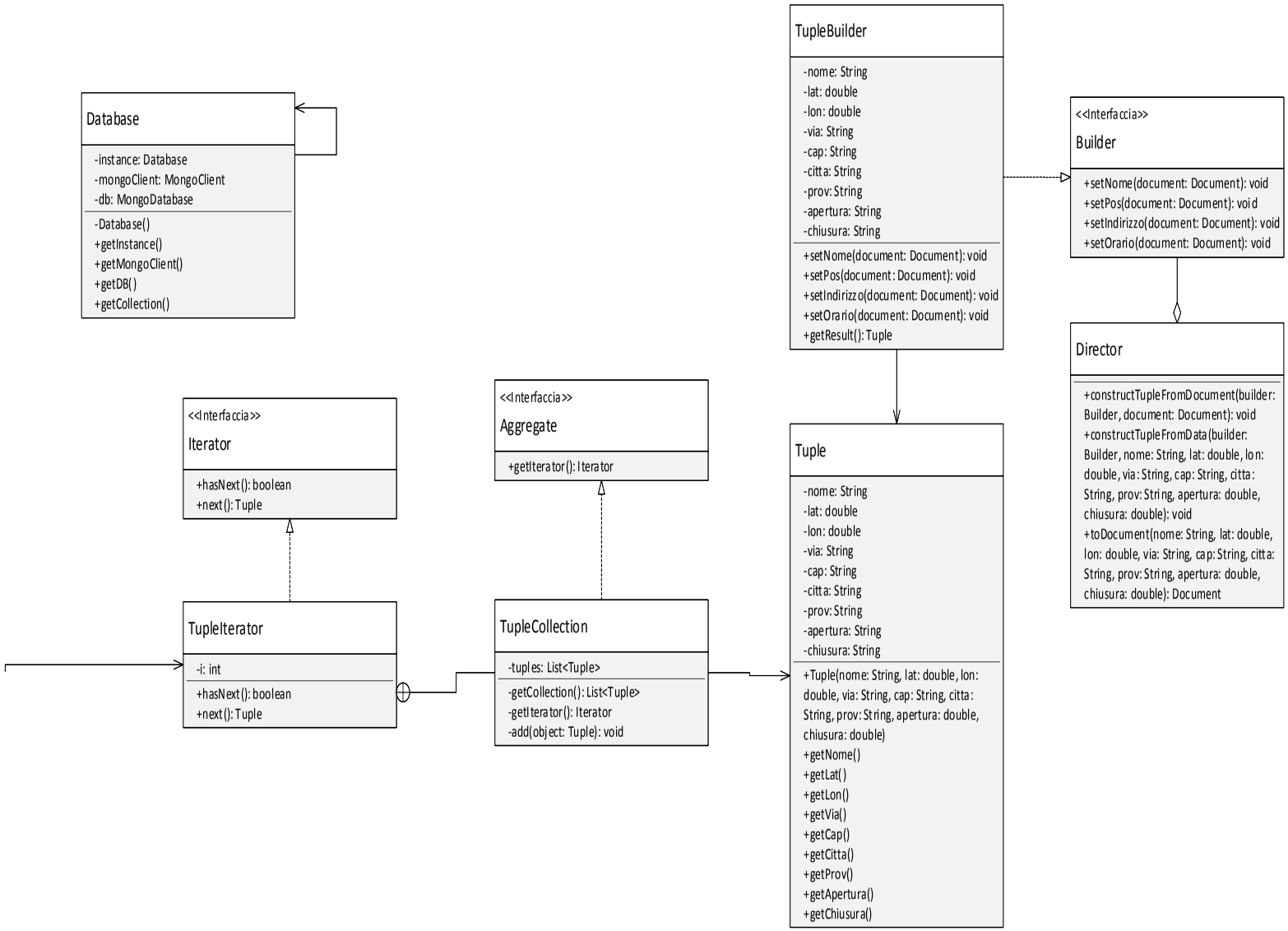
Capitolo 2

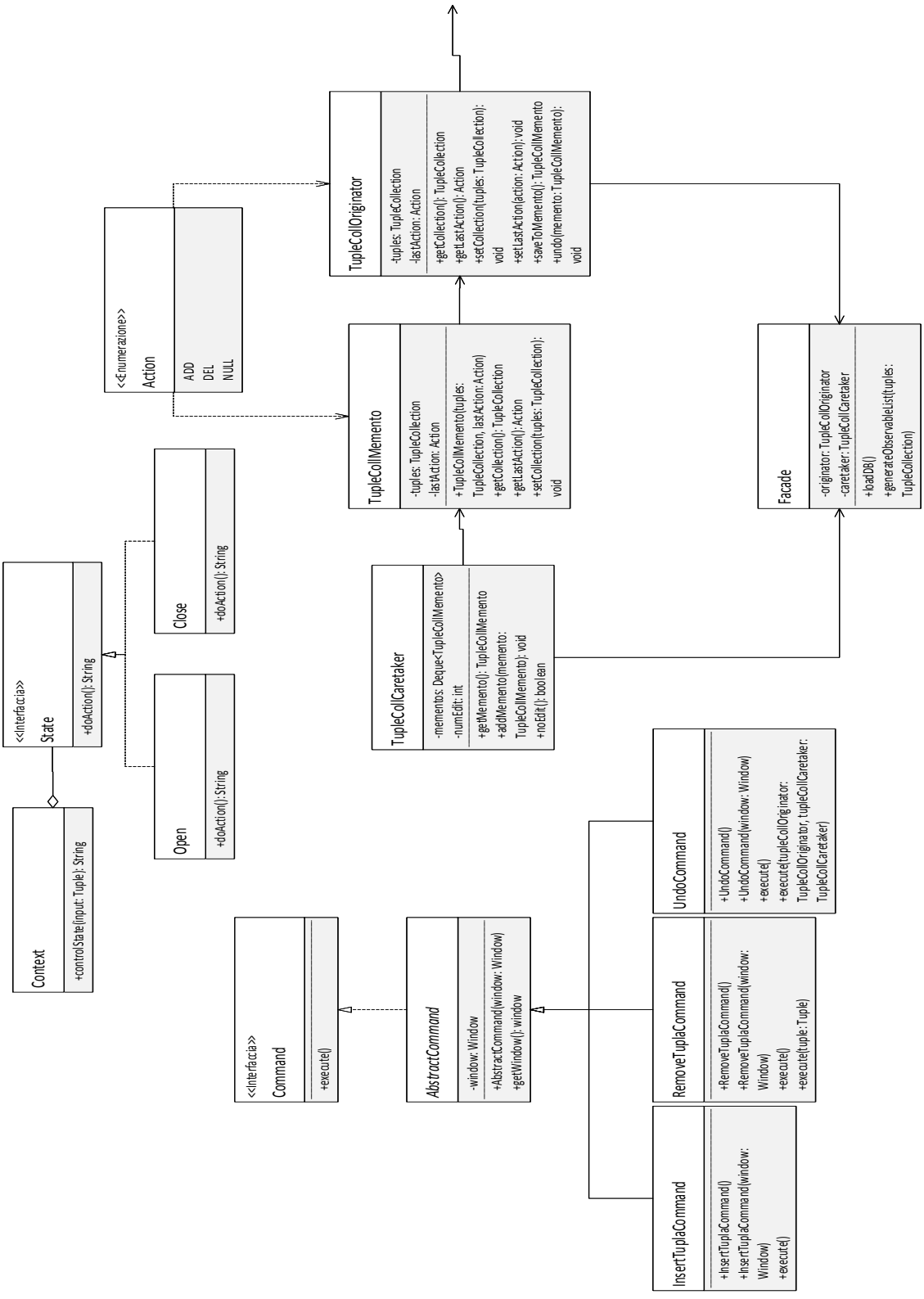
Diagramma UML delle classi

Le classi concrete implementate sono:

- Database
- Director
- Builder
- TupleBuilder
- Tuple
- TupleCollection
- TupleIterator
- TupleCollOriginator
- TupleCollMemento
- TupleCollCaretaker
- Facade
- Context
- Open
- Close
- InsertTuplaCommand
- RemoveTuplaCommand
- UndoCommand

Di seguito viene illustrato il diagramma UML delle classi implementate. Successivamente vengono approfondite alcune delle classi più importanti.


Database
-instance: Database
-mongoClient: MongoClient
-db: MongoDatabase
-Database()
+getInstance()
+getMongoClient()
+getDB()
+getCollection()
TupleBuilder
-nome: String
-lat: double
-lon: double
-via: String
-cap: String
-citta: String
-prov: String
-apertura: String
-chiusura: String
+setNome(document: Document): void
+setPos(document: Document): void
+setIndirizzo(document: Document): void
+setOrario(document: Document): void
+getResult(): Tuple
<<Interfaccia>>
Builder
+setNome(document: Document): void
+setPos(document: Document): void
+setIndirizzo(document: Document): void
+setOrario(document: Document): void
Director
+constructTupleFromDocument(builder: Builder, document: Document): void
+constructTupleFromData(builder: Builder, nome: String, lat: double, lon: double, via: String, cap: String, citta: String, prov: String, apertura: double, chiusura: double): void
+toDocument(nome: String, lat: double, lon: double, via: String, cap: String, citta: String, prov: String, apertura: double, chiusura: double): Document
<<Interfaccia>>
Iterator
+hasNext(): boolean
+next(): Tuple
<<Interfaccia>>
Aggregate
+getIterator(): Iterator
Tuple
-nome: String
-lat: double
-lon: double
-via: String
-cap: String
-citta: String
-prov: String
-apertura: String
-chiusura: String
+Tuple(nome: String, lat: double, lon: double, via: String, cap: String, citta: String, prov: String, apertura: double, chiusura: double)
+getNome()
+getLat()
+getLon()
+getVia()
+getCap()
+getCitta()
+getProv()
+getApertura()
+getChiusura()
TupleIterator
-i: int
+hasNext(): boolean
+next(): Tuple
TupleCollection
-tuples: List<Tuple>
-getCollection(): List<Tuple>
-getIterator(): Iterator
-add(object: Tuple): void

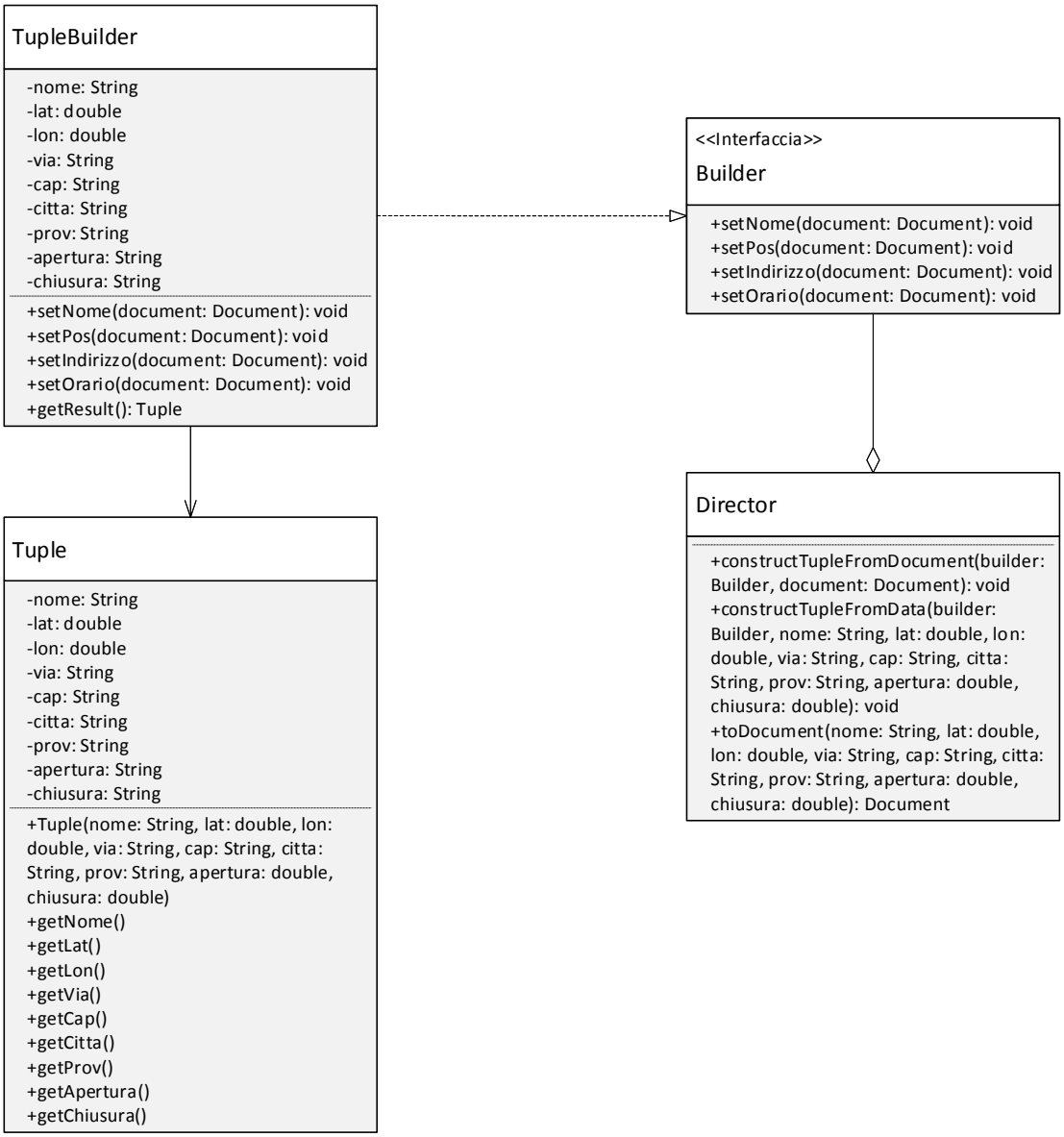


2.1 UML del costruttore di tuple

Attraverso il creational pattern *Builder* è stato implementato un modo per "costruire" le tuple a partire da diverse tipologie di dati in input. Lo scopo del pattern

consiste proprio nel costruire oggetti complessi passo dopo passo.

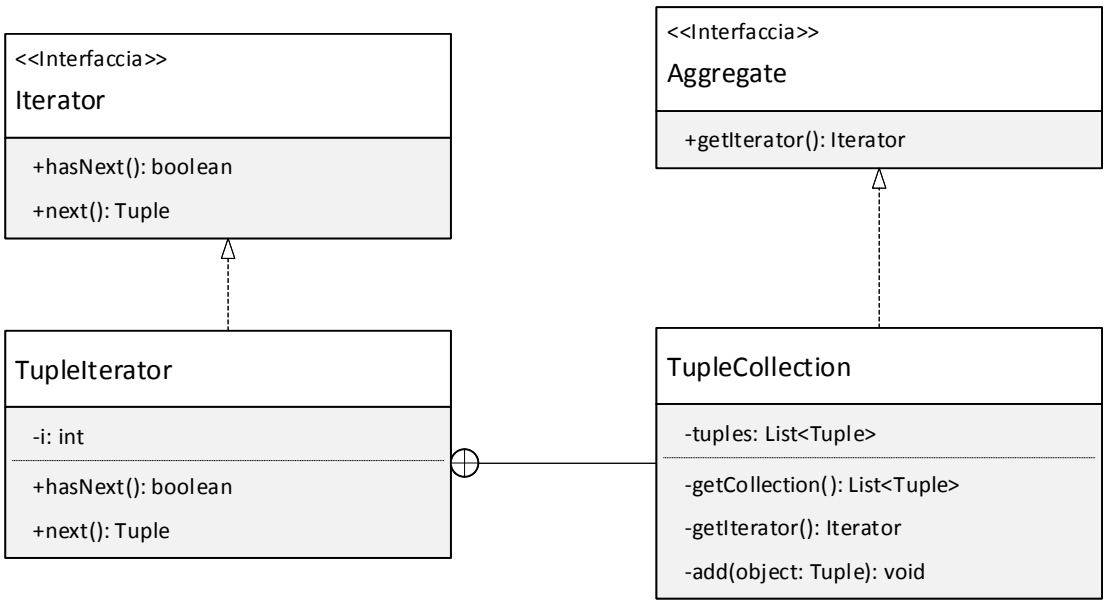
Il costruttore può prendere in input diversi tipi di dati: un oggetto *Document*[1] oppure i singoli dati che compongono una tupla. Un oggetto di tipo *Tuple* corrisponde ad un segnaposto sulla mappa e contiene tutti i dettagli di esso, come le coordinate, il nome della farmacia di riferimento, l'ubicazione e gli orari di servizio.



2.2 UML della collezione di tuple

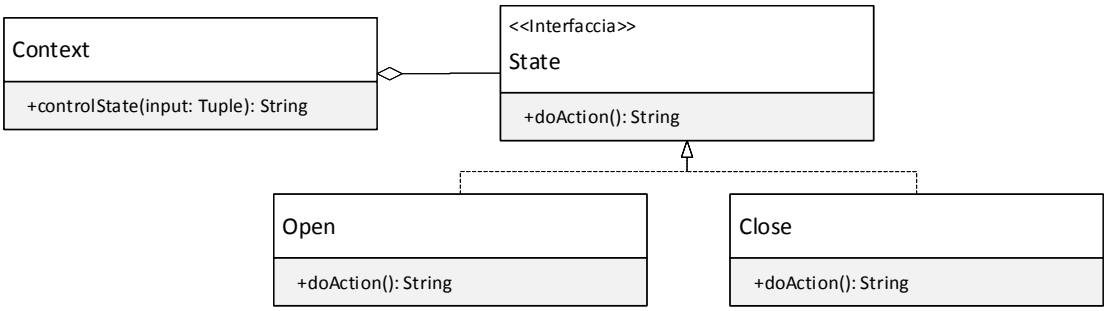
Al fine di agevolare l'accesso alle tuple e nasconderne i dettagli implementativi è stata scritta una classe che contenesse un insieme di tuple. La *TupleCollection* prevede una lista di tuple e una serie di metodi per l'accesso ad essa. All'interno della classe è stato sviluppato il behavioral pattern *Iterator*, che consente di accedere alla collezione di tuple senza conoscerne i dettagli che costituiscono la struttura dati.

[1]Oggetto implementato dal package dei driver MongoDB.



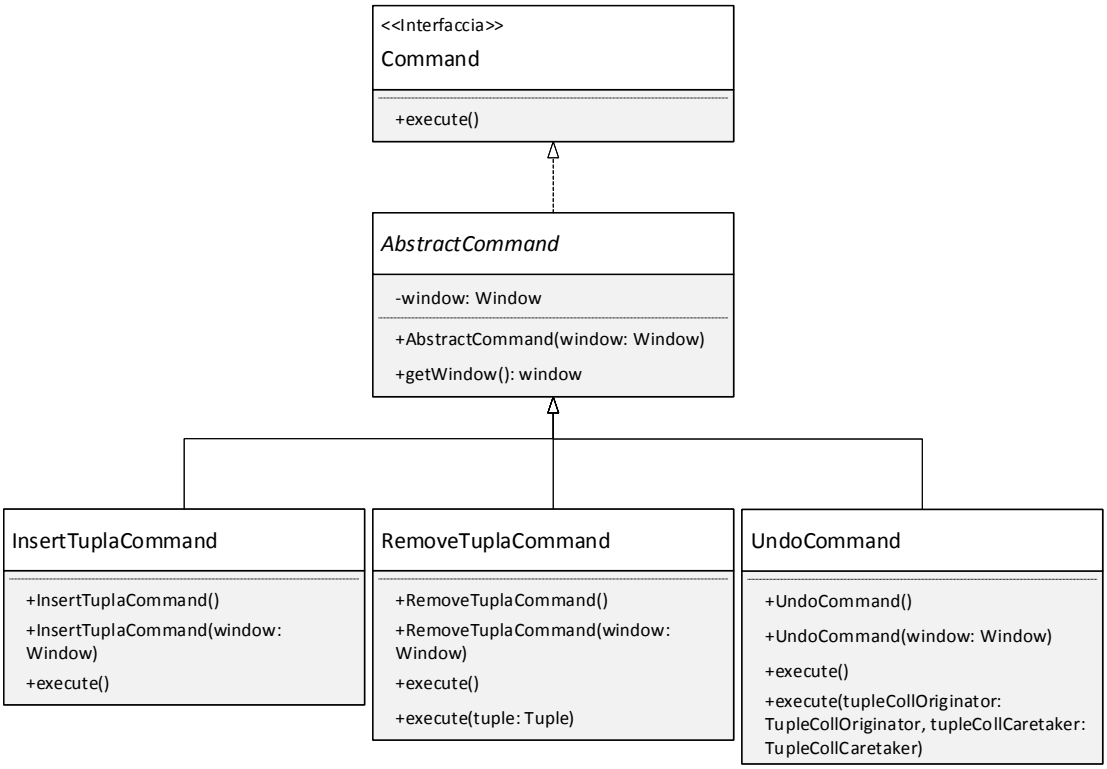
2.3 UML della lettura degli orari di servizio

Attraverso il behavorial pattern *State* che consente a un oggetto di modificare il suo comportamento quando cambia il suo stato interno è stato possibile implementare un meccanismo automatizzato che consenta di rilevare automaticamente lo stato di servizio di una farmacia di riferimento.



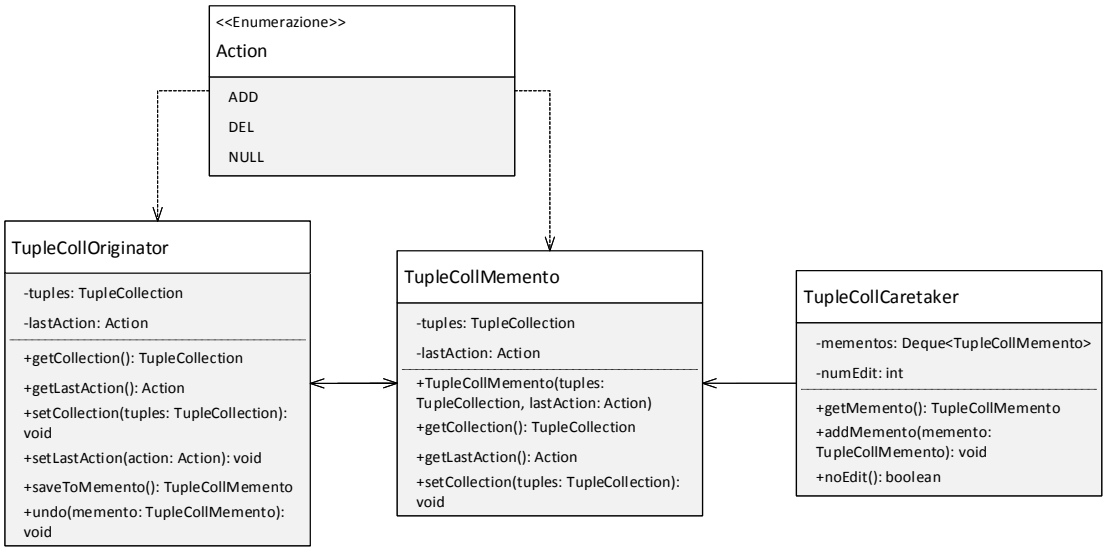
2.4 UML della struttura dei comandi

Grazie al beheavioral pattern *Command* sono stati implementati i tre comandi principali dell'applicativo back-office, trasformandoli in oggetti autonomi e indipendenti dall'interfaccia grafica. Le operazioni di inserimento, cancellazione e "annulla" sono rispettivamente tre classi che ne implementano una astratta. In particolare l'annullamento delle operazioni ingloba un ulteriore pattern approfondito successivamente.



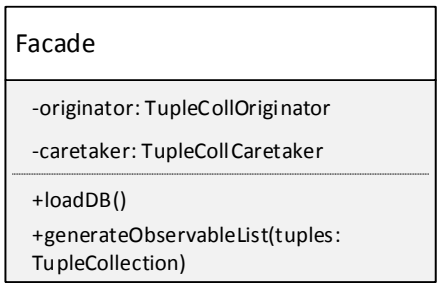
2.5 UML dell'annullamento delle operazioni

Una delle funzioni principali dell'applicativo consiste nell'annullamento delle modifiche effettuate alla base di dati, tornando di volta in volta "indietro". Grazie al behavioral pattern *Memento* la collezione di tuple viene incapsulata in una classe nella quale è possibile salvare degli "snapshot" della collezione e tenere conto dell'ultima operazione effettuata. Una classe custode si occupa di mantenere una copia di tutti gli stati dell'oggetto di tipo *TupleCollection*. Grazie alla classe *UndoCommand*, facente parte del pattern *Command* è stato possibile implementare con facilità tale operazione, isolando il contesto delle classi dall'operazione effettiva.



2.6 UML dell'accesso semplificato alle classi

La catena di incapsulamenti delle classi rende inevitabilmente più difficoltosa la comprensione dei suoi funzionamenti. Al fine di rendere il codice più leggibile e ridurre al minimo il numero di istanziamenti di vari oggetti si è optato per l'implementazione di uno structural pattern, il *Facade*. In questo caso istanziare la classe di facciata permette di avere già pronta una collezione di tuple ed assicurarsi che il "custode" memorizzi ogni modifica applicata. Un ulteriore metodo consente di caricare tutti i documenti contenuti nel database remoto e di convertirli in oggetti di tipo *Tuple*.



Capitolo 3

Dettagli implementativi

Di seguito sono riportate alcune porzioni rilevanti del codice sviluppato.

3.1 Inserimento di una farmacia

L'applicativo permette l'inserimento di una tupla mediante un form ed effettua l'aggiornamento del database remoto. Il funzionamento consiste nei seguenti passaggi:

1. Legge i dati dal form
2. Converte le stringhe in un oggetto tupla
3. Invia i dati al server remoto sotto forma di Document
4. Sincronizza la base di dati tra locale e remoto

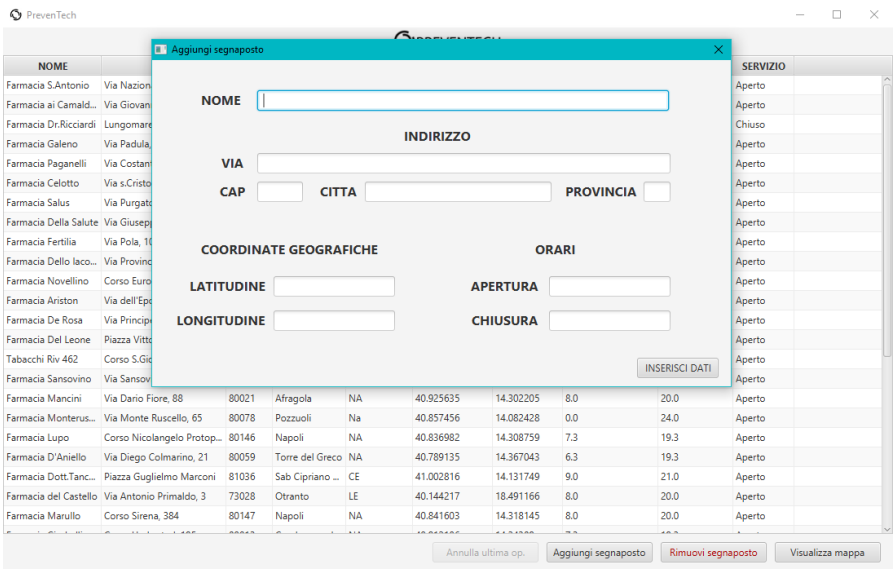


Figura 3.1: Form di inserimento

```
public Optional<Boolean> execute() {
    FXMLLoader fxmlLoader =
```

```
            new FXMLLoader(getClass().getResource("FXMLInsertTuplaForm.fxml"));
        Parent root;
        try {
            root = fxmlLoader.load();
        } catch (IOException ex) {
            Logger.getLogger(InsertTuplaCommand.class.getName())
                .log(Level.SEVERE, null, ex);
            return null;
        }
        Stage stage = new Stage();
        stage.initModality(Modality.APPLICATION_MODAL);
        stage.setOpacity(1);
        stage.setTitle("Aggiungi segnaposto");
        stage.setScene(new Scene(root));
        stage.setResizable(false);
        stage.showAndWait();
        return null;
    }

    private void insertTupla(ActionEvent event) throws IOException {
        Director director = new Director();
        TupleBuilder builder = new TupleBuilder();
        director.constructTupleFromData(builder, nome.getText(),
            Double.parseDouble(latitudine.getText()),
            Double.parseDouble(longitudine.getText()),
            via.getText(), cap.getText(), citta.getText(), prov.getText(),
            Double.parseDouble(apertura.getText()),
            Double.parseDouble(chiusura.getText()));
        builder.getResult().inserisciInDB();
    }
```

3.2 Cancellazione di una farmacia

In maniera simile all'inserimento, la cancellazione prevede la lettura di dati, l'invio della richiesta al database remoto e la sincronizzazione tra locale e remoto. La scelta avviene mediante interfaccia grafica, dove basta selezionare la riga desiderata. I passaggi sono i seguenti:

1. Legge i dati dal form
2. Converte le stringhe in un oggetto tupla
3. Invia i dati al server remoto sotto forma di Document
4. Sincronizza la base di dati tra locale e remoto

```
    private void rimuoviSegnaposto(MouseEvent event) throws IOException {
        instance.getOriginator().setLastAction(Action.DEL);
        instance.getCaretaker().
            addMemento(instance.getOriginator().saveToMemento());
```

```
    Director director = new Director();
    TupleBuilder builder = new TupleBuilder();
    String nome = table.getSelectionModel().getSelectedItem().getNome();
    double lat = table.getSelectionModel().getSelectedItem().getLat();
    double lon = table.getSelectionModel().getSelectedItem().getLon();
    String via = table.getSelectionModel().getSelectedItem().getVia();
    String cap = table.getSelectionModel().getSelectedItem().getCap();
    String citta = table.getSelectionModel().getSelectedItem().getCitta();
    String prov = table.getSelectionModel().getSelectedItem().getProv();
    double apertura = table.getSelectionModel().
        getSelectedItem().getApertura();
    double chiusura = table.getSelectionModel().
        getSelectedItem().getChiusura();
    director.constructTupleFromData(builder, nome, lat, lon,
        via, cap, citta, prov, apertura, chiusura);
    new RemoveTuplaCommand().execute(builder.getResult());
    instance.getOriginator().setCollection(Facade.loadDB());
    table.setItems(generateObservableList(instance.
        getOriginator().getCollection()));
    if (undoButton.isDisabled())
    undoButton.setDisable(false);
}

public void execute(Tuple tuple) {
    tuple.rimuoviDalDB();
}
```

3.3 Annullamento di una operazione

Ogni operazione effettuata sulla base di dati viene memorizzata in una pila, dalla quale sarà possibile di volta in volta estrarre l'ultimo snapshot della collezione di tuple e ripristinare il database remoto allo stato precedente. Quando si esegue una operazione di inserimento o cancellazione, viene creata una copia dello stato attuale del database e viene "marchiata" con l'operazione che viene applicata alla base di dati, questo consente di annullare le operazioni applicando l'opzione "opposta" a quella effettuata: per annullare un inserimento si elimina l'ultima tupla inserita, viceversa per annullare l'eliminazione di una tupla si inserisce la tupla che nello stato precedente della TupleCollection non risulta presente nel database remoto, quindi la inserisce.

```
public void execute(TupleCollOriginator tupleCollOriginator,
                    TupleCollCaretaker tupleCollCaretaker) {
    switch(tupleCollOriginator.getLastAction()) {
        case ADD:
            tupleCollOriginator.getCollection()
                .getCollection().get(tupleCollOriginator
                .getCollection().getCollection().size()-1).rimuoviDalDB();
            tupleCollOriginator.undo(tupleCollCaretaker.getMemento());
```

```
                break;
            case DEL:
                tupleCollOriginator.undo(tupleCollCaretaker.getMemento());
                Tuple tuple;
                Iterator iter = tupleCollOriginator.getCollection().getIterator();
                do {
                    tuple = iter.next();
                } while (iter.hasNext() && tuple.isInDB());
                tuple.inserisciInDB();
                break;
        }
    }
```

Capitolo 4

Istruzioni per l'esecuzione

Il codice è rilasciato sotto forma di due progetti, entrambi necessari al funzionamento. È possibile compilare il codice da un qualsiasi IDE, una volta generati i due eseguibili si possono lanciare direttamente. I due progetti hanno le seguenti denominazioni:

- `PrevenTechFX`: applicativo back-office con interfaccia grafica JavaFX
- `PrevenTechServ`: servlet del sito web prevenTech, porting in Java del progetto originario in Flask-Python

Per il corretto funzionamento della servlet[2] è necessario impostare la porta su `8180`.

[2]Durante lo sviluppo è stato utilizzato il software **Apache Tomcat**, versione 9.0.