Studente: Gabriele Romanato
Matricola: 258820

Progetto finale – Traccia 1

CORSO DI OBJECT ORIENTED SOFTWARE DESIGN CORSO DI LAUREA IN INFORMATICA UNIVERSITA' DEGLI STUDI DELL'AQUILA A.A. 2019/2020

GitHub: https://github.com/gabrieleromanato/univaq-pedaggio-autostradale

Nota: Gli screenshot del codice presenti nel documento potrebbero non essere aggiornati con le recenti modifiche. Si faccia riferimento alla repository GitHub.

1. Creazione della base di dati

Poiché l'amministratore del sistema dovrà poter aggiornare l'elenco delle autostrade, si è ritenuto opportuno partire dall'elenco di autostrade italiane ufficialmente attive.

Collegandosi all'indirizzo http://www.autostrade.it/autostrade-gis/percorso.do si è innanzitutto provveduto ad estrarre dalla pagina web l'elenco delle autostrade con il relativo codice identificativo sfruttando la console JavaScript del browser.

```
let csv = 'codice,nome\n';
document.querySelector('select[name=cod]').querySelectorAll('option').forEach(opt => {
    csv += opt.getAttribute('value') + ',' + opt.innerText + '\n';
});
console.log(csv);
```

Quindi, poiché il progetto prevede l'uso di JDBC, si è provveduto a creare un database MySQL locale in cui si è importato il file CSV creato copiando l'output della console JavaScript del browser.

La tabella **autostrade** contiene tre colonne:

Nome	Tipo di dati	Lunghezza / Length
id	INT	11
nome	VARCHAR	255
codice	VARCHAR	3

Collegandoci all'indirizzo https://www.autostrade.it/autostrade-gis/rete.do?op=caselli possiamo estrarre l'elenco dei caselli con i relativi dati utilizzando su ciascuna tabella delle pagine di dettaglio il seguente codice JavaScript:

```
function getData() {
    let parts = location.href.split('&');
    let id = parts[1].split('=')[1];
    let csv = '';
    const rows = document.querySelectorAll('table.elenco tr');
    rows.forEach(row => {
        const cells = row.querySelectorAll('td');
        let km = cells[0];
        let name = cells[1];
        let code = cells[2];

        if(km && name && code) {

        if(/^\d+$/.test(code.innerHTML.trim())) {
            csv += id + ',' + km.innerHTML + ',' + name.innerHTML + ',' + code.innerHTML + '\n';
        }

        }

    });
    console.log(csv);
};

getData();
```

Quindi possiamo importare il file CSV creato nella tabella **caselli** che avrà la seguente struttura:

Nome	Tipo di dati	Lunghezza / Length
id	INT	11
autostrada	VARCHAR	3
progressiva_km	DECIMAL	10,2
nome	VARCHAR	255
codice	VARCHAR	255

Il campo `autostrada` corrisponde al codice di un'autostrada precedentemente salvato nella tabella `autostrade`. In questo modo effettuiamo il collegamento tra i due dati.

Il campo `progressiva_km` indica il numero di chilometri associati a ciascun casello. Con questo dato è possibile calcolare il pedaggio per ciascun casello.

Dato che il progetto prevede anche il pagamento del pedaggio, ne consegue che il database dovrà contenere una tabella per la registrazione dei pagamenti. Questa tabella, **pagamenti**, potrà avere la seguente struttura.

Nome	Tipo di dati	Lunghezza / Length
id	INT	11
id_veicolo	VARCHAR	255
importo	DECIMAL	10,2
orario	DATETIME	
tipologia	VARCHAR	255

id_veicolo potrebbe essere rappresentato dalla targa del veicolo. In questo modo in un real case scenario il database centrale potrebbe dialogare con il database della motorizzazione per ulteriori implementazioni tramite API.

id nel nostro database è di tipo `AUTO_INCREMENT` [1] e quindi diventa facoltativo nel design delle classi che seguono.

1.1 OOP per la base di dati

Come premessa è utile ricordare due dei principi espressi in *Design Patterns. Elements of Reusable Object Oriented Software* (1995):

1. *Program to an interface, not an implementation.*
2. *Favor object composition over class inheritance.*

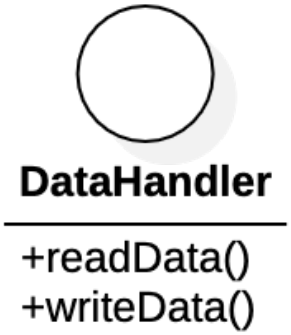
Il primo principio è immediatamente applicabile alla gestione dei dati. Se in futuro infatti volessimo sostituire una soluzione con database con una soluzione di diverso tipo (REST API, flat file ecc.), abbiamo bisogno di un'interfaccia che fornisca la necessaria genericità per sostituire un'implementazione con un'altra.

Questa interfaccia dovrà fornire due metodi:

1. Un metodo per la lettura dei dati.
2. Un metodo per la scrittura dei dati.

[1] https://dev.mysql.com/doc/refman/8.0/en/example-auto-increment.html

Possiamo chiamare questa interfaccia `DataHandler` e assegnarle la seguente struttura.

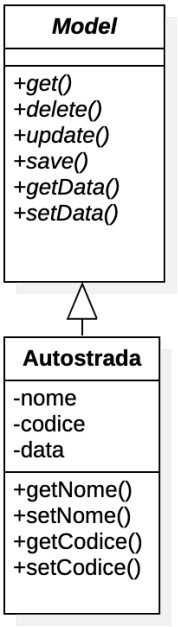



```java
package interfaces;

import java.util.ArrayList;

public interface DataHandler {

    ArrayList<?> readData();

    boolean writeData();

}
```

Nell'ottica di seguire il pattern MVC, dobbiamo definire dei modelli (Model) per i dati che andremo a gestire.

Per procedere dobbiamo creare una classe astratta che rappresenti un modello generico di dati e le operazioni disponibili su di essi.

Il primo dato, gestibile dall'amministratore del sistema, verrà rappresentato dalla classe `Autostrada` che estende la classe astratta `Model`.



La proprietà `data` rappresenta una versione dei dati delle classi figlie che può essere usata in una query SQL o comunque in modo da consentire la serialisation, come ad esempio `HashMap`.

```
package models;

import java.util.HashMap;

public abstract class Model {

    public abstract HashMap<?, ?> get();
    public abstract boolean save();
    public abstract boolean update();
    public abstract boolean delete();


}
```


```java
package models;

import java.util.HashMap;

public class Autostrada extends Model {

    private String nome;
    private String codice;
    private HashMap<?,?> data;

    public Autostrada() {
        this("A0", "A00");
    }


    public Autostrada(String nome, String codice) {
        this.nome = nome;
        this.codice = codice;
    }

    @Override
    public HashMap<?, ?> get() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public boolean save() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean update() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean delete() {
        // TODO Auto-generated method stub
        return false;
    }

    public void setData(HashMap<?,?> data) {
        this.data = data;
    }

    public HashMap<?,?> getData() {
        return null;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getCodice() {
        return codice;
    }

    public void setCodice(String codice) {
        this.codice = codice;
    }

}
```

Il modello per i pagamenti, la classe `Pagamento`, seguirà il modello ereditario della classe `Autostrada` e avrà la seguente implementazione iniziale.

```java
package models;

import java.util.Date;
import java.util.HashMap;

public class Pagamento extends Model {

    private String idVeicolo;
    private double importo;
    private Date orario;
    private String tipologia;
    private HashMap<?, ?> data;

    public Pagamento() {
        this("AZ1234", 9.99, "contanti");
    }

    public Pagamento(String idVeicolo, double importo, String tipologia) {
        this.idVeicolo = idVeicolo;
        this.importo = importo;
        this.orario = new Date();
        this.tipologia = tipologia;
    }

    public HashMap<?, ?> getData() {
        return data;
    }

    public void setData(HashMap<?, ?> data) {
        this.data = data;
    }

    public String getIdVeicolo() {
        return idVeicolo;
    }

    public void setIdVeicolo(String idVeicolo) {
        this.idVeicolo = idVeicolo;
    }

    public double getImporto() {
        return importo;
    }

    public void setImporto(double importo) {
        this.importo = importo;
    }

    public Date getOrario() {
        return orario;
    }

    public void setOrario(Date orario) {
        this.orario = orario;
    }

    public String getTipologia() {
        return tipologia;
    }

    public void setTipologia(String tipologia) {
        this.tipologia = tipologia;
    }

    @Override
    public HashMap<?, ?> get() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public boolean save() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean update() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean delete() {
        // TODO Auto-generated method stub
        return false;
    }

}
```

Poiché nel progetto esiste già una classe chiamata `Casello`, il modello che gestirà il tipo di dati casello si chiamerà `CaselloModel` in modo da evitare ambiguità nello sviluppo e aiutare la funzionalità di suggerimento dell'IDE. Avrà la seguente struttura:

```java
import java.util.HashMap;

public class CaselloModel extends Model {

    private String autostrada;
    private double progressivaKm;
    private String nome;
    private String codice;
    private HashMap<?,?> data;

    public CaselloModel() {
        this("A00", 1.00, "Casello", "000");
    }


    public CaselloModel(String autostrada, double progressivaKm, String nome, String codice) {
        this.autostrada = autostrada;
        this.progressivaKm = progressivaKm;
        this.nome = nome;
        this.codice = codice;
    }

    public String getAutostrada() {
        return autostrada;
    }

    public void setAutostrada(String autostrada) {
        this.autostrada = autostrada;
    }

    public double getProgressivaKm() {
        return progressivaKm;
    }

    public void setProgressivaKm(double progressivaKm) {
        this.progressivaKm = progressivaKm;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getCodice() {
        return codice;
    }

    public void setCodice(String codice) {
        this.codice = codice;
    }

    public HashMap<?, ?> getData() {
        return data;
    }

    public void setData(HashMap<?, ?> data) {
        this.data = data;
    }

    @Override
    public HashMap<?, ?> get() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public boolean save() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean update() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean delete() {
        // TODO Auto-generated method stub
        return false;
    }

}
```

Abbiamo definito in precedenza l'interfaccia `DataHandler`. Ora dobbiamo creare una classe che la implementi e che possa essere utilizzata dai model. Chiameremo questa classe `Database`.

Nel costruttore creeremo la connessione al database MySQL utilizzando il driver ufficiale.

```
public Database() {
    try {

        this.conn = DriverManager.getConnection("jdbc:mysql://localhost/pedaggioautostradale?user=pedaggio&password=1234&useSSL=false&serverTimezone=Europe/Rome");
    } catch (SQLException ex) {
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    }
}
```

```
public ArrayList<HashMap<String, String>> readData( String query ) {
        Statement stmt = null;
        ResultSet rs = null;
        ArrayList<HashMap<String,String>> data = new ArrayList<HashMap<String,String>>();

        try {
            stmt = this.conn.createStatement();
            rs = stmt.executeQuery(query);
            ArrayList<String> info = new ArrayList<String>();

            ResultSetMetaData metadata = rs.getMetaData();

            int columnCount = metadata.getColumnCount();

            for (int i = 1; i <= columnCount; i++ ) {

                String columnName = metadata.getColumnName(i);
                String columnType = metadata.getColumnTypeName(i);

                info.add(columnName + ":" + columnType);

                
            }

            if (stmt.execute(query)) {
                rs = stmt.getResultSet();

                while(rs.next()) {

                    HashMap<String, String> datum = new HashMap<String, String>();

                    for(String part: info) {
                        String[] parts = part.split(":");
                        String name = parts[0];
                        String type = parts[1];

                        switch(type.toLowerCase()) {
                        case "int":
                            datum.put(name, Integer.toString(rs.getInt(name)));
                            break;
                        case "varchar":
                            datum.put(name, rs.getString(name));
                            break;
                        case "decimal":
                            datum.put(name, Double.toString(rs.getDouble(name)));
                            break;
                        case "datetime":
                            Timestamp timestamp = rs.getTimestamp(name);
                            String pattern = "yyyy-MM-dd HH:mm:ss";
                            SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern);
                            datum.put(name, simpleDateFormat.format(timestamp.getTime()));
                            break;
                            default:
                                break;
                        }
                    }

                    data.add(datum);

                }

                return data;

            }


        } catch(SQLException ex) {
            System.out.println("SQLException: " + ex.getMessage());
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("VendorError: " + ex.getErrorCode());
        } finally {
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException sqlEx) { }
            }

            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException sqlEx) { }
            }
        }
        return data;
}
```

Il metodo `readData()` ,da implementare per soddisfare il contratto con l'interfaccia `DataHandler`, compie diverse operazioni sui risultati di una query SQL. La prima consiste nell'estrarre i nomi delle colonne e il tipo di dati associati ad esse. Con queste informazioni si andranno a reperire i dati usando i metodi appropriati della classe `ResultSet` e uniformandoli nel formato `String`. Queste operazioni si sono rese necessarie in quanto non è possibile riutilizzare un oggetto `ResultSet` dopo che la transazione viene chiusa.

Il metodo `writeData()` esegue una query di aggiornamento sul database.

```
public boolean writeData(String query) {

      try {
            Statement st = this.conn.createStatement();
            st.executeUpdate(query);
            this.conn.close();
            return true;
         } catch(SQLException ex) {
            try {
                this.conn.close();
            } catch(SQLException e) {
                System.out.println(e.getErrorCode());
            }
            return false;
         }

}
```

Questa classe verrà usata dai model sfruttando il principio della composition [2] e troverà un'applicazione nei metodi CRUD dei model.

[2] *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 20

```java
private Database db;

public Autostrada() {
    this("A0", "A00");
}


public Autostrada(String nome, String codice) {
    this.nome = nome;
    this.codice = codice;
    this.db = new Database();
}
```