Университет ИТМО
Кафедра вычислительной техники

Лабораторная работа №7
по дисциплине «Программирование»
Вариант: 961.

Выполнил: Юсюмбели Владислав Иванович
Проверил: Письмак Алексей Евгеньевич

Санкт—Петербург – 2017г.

Разделить программу из лабораторной работы №6 на клиентский и серверный модули. Клиентский модуль должен обеспечивать взаимодействие с пользователем с помощью графического интерфейса. Серверный модуль должен реализовывать все действия с коллекцией, возвращая клиенту данные для отображения. Данные для коллекции должны храниться в базе данных PostgreSQL. Объекты должны передаваться в сериализованном виде.

Сервер должен поддерживать работу с несколькими клиентами, блокируя одновременные запросы на изменение данных. В случае, если клиент пытается выполнить операцию с неактуальными данными, сервер должен извещать об этом клиента. При этом клиент должен получить от сервера актуальные данные и обновить их у себя.

Получившаяся в итоге программа должна удовлетворять следующим требованиям:

1. Обмен данными между клиентом и сервером должен осуществляться по протоколу UDP.
2. При этом сервер должен использовать сетевой канал а клиент - датаграммы.
3. Для соединения с базой данных использовать org.postgresql.ds.PGConnectionPoolDataSource.
4. Имя пользователя и пароль для соединения с базой задавать в аргументах метода getPooledConnection().
5. Для получения результатов запроса использовать java.sql.ResultSet.
6. Групповые операции удаления и вставки данных должны быть реализованы с использованием транзакций.
7. Одиночные операции модификации данных должны быть реализованы с использованием метода PreparedStatement.executeUpdate().

Исходный код программы:

class ServerLoader

```
package DataFromClitent;

import GUI.Button;
import connectDB.MessageToClient;
import connectDB.WorkWithDB;
import old.school.People;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.net.UnknownHostException;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;
import java.util.Map;
import java.util.NoSuchElementException;
import java.util.concurrent.ConcurrentHashMap;

/**
 * Created by slavik on 01.05.17.
 */
public class ServerLoader {
    private static ByteArrayOutputStream oldData = new ByteArrayOutputStream();
    private static ByteArrayOutputStream newData = new ByteArrayOutputStream();
    private static Button button;
    private static Data typeOfData = Data.OLD;

    public static void main(String[] args) throws UnknownHostException {
        SocketAddress inetSocketAddress = new InetSocketAddress(InetAddress.getLocalHost(), 7007);
        while (true) {
            try (DatagramChannel serverSocket = DatagramChannel.open().bind(inetSocketAddress)) {
                System.out.println(serverSocket);
                ByteBuffer dataFromClient = ByteBuffer.allocate(8 * 1024);
                while (true) {
                    SocketAddress socketAddress = serverSocket.receive(dataFromClient);

                    String msgFromClient = new String(dataFromClient.array(), 0, dataFromClient.position());


                    MessageToClient messageToClient = analysisMsgFromClient(msgFromClient, dataFromClient);
                    if (messageToClient != null) {
                        messageToClient.sendData(serverSocket, socketAddress);
                    }

                    dataFromClient.clear();
                }


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

    /*
    oldData, NEW, newData, BUTTON, BUTTON, END
     */

    private static MessageToClient analysisMsgFromClient(String msgFromClient, ByteBuffer dataFromClient) throws IOException {

        if (msgFromClient.equals("END")) {
            typeOfData = Data.OLD;
            WorkWithDB workWithDB = new WorkWithDB(oldData, newData, button);
            MessageToClient messageToClient =workWithDB.executeCommand();
            newData.reset();
            oldData.reset();
            return messageToClient;
        }

        try {
            switch (typeOfData) {
                case NEW:
                    newData.write(dataFromClient.array());
                    break;
                case OLD:
                    oldData.write(dataFromClient.array());
                    break;
                case BUTTON:
                    button = Button.valueOf(msgFromClient);
                    break;
            }
        } catch (IllegalArgumentException e) {
            //do nothing
        }

        try {
            typeOfData = Data.valueOf(msgFromClient);
        } catch (IllegalArgumentException e) {
//            System.out.println(e.getMessage());
        }

        return null;
    }
}
```

enum Data

```
package DataFromClitent;

/**
 * Created by slavik on 01.05.17.
 */
public enum Data {
    NEW,
    OLD,
    BUTTON;
}
```

enum Data

```
package DataFromClitent;

/**
 * Created by slavik on 01.05.17.
 */
public enum Command {
    UPDATE,
    REMOVE,
    INSERT
}
```

enum Button

```
package GUI;

import com.sun.org.apache.regexp.internal.RE;
import com.sun.xml.internal.ws.api.ha.StickyFeature;
import old.school.Man;
import org.postgresql.util.PSQLException;
import org.postgresql.util.ServerErrorMessage;

import javax.mail.*;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.sql.*;
import java.util.*;

import static java.sql.ResultSet.CONCUR_UPDATABLE;
import static java.sql.ResultSet.TYPE_FORWARD_ONLY;

/**
 * Created by slavik on 03.05.17.
 */
public enum Button {
    /**
     * Команда: remove_greater_key.
     * Удаляет из коллекции все элементы, ключ которых превышает заданный.
     *
     * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
     * @version 3.0
     */
    REMOVE_GREATER_KEY {
        private int updateRow;

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            msgToClient = "Objects removed";
            try {
                connection.setAutoCommit(false);
                Statement statement = connection.createStatement();

                int key = Integer.parseInt(newData.entrySet().iterator().next().getKey());

                updateRow = statement.executeUpdate("DELETE FROM people WHERE id> " + key);
                connection.commit();
            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            } catch (IllegalArgumentException e) {
                msgToClient = "Key is not correct";
            }
            return updateRow;
        }
    },


    /**
     * Команда remove_lower.
     * Удаляет из коллекции все элементы, ключ которых меньше, чем заданный.
     *
     * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
     * @version 1.0
     */
    REMOVE_LOWER_KEY {
        private int updateRow;

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            msgToClient = "Objects removed";
            try {
                connection.setAutoCommit(false);
                Statement statement = connection.createStatement();

                int key = Integer.parseInt(newData.entrySet().iterator().next().getKey());

                updateRow = statement.executeUpdate("DELETE FROM people WHERE id < " + key);
                connection.commit();
            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            } catch (IllegalArgumentException e) {
                msgToClient = "Key is not correct";
            }
            return updateRow;
        }
    },


    /**
     * Команда remove.
     * Удаляет элемент из коллекции по его ключу.
     *
     * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
     * @version 3.0
     */
    REMOVE_WITH_KEY {
        private int updateRow;

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            msgToClient = "Objects removed";
            try {
                connection.setAutoCommit(false);
                Statement statement = connection.createStatement();

                int key = Integer.parseInt(newData.entrySet().iterator().next().getKey());

                updateRow = statement.executeUpdate("DELETE FROM people WHERE id = " + key);
                connection.commit();
            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            } catch (IllegalArgumentException e) {
                msgToClient = "Key is not correct";
            }
            return updateRow;
        }
    },
    /**
     * Команда remove_greater.
     * Удаляет из коллекции все элементы, превышающие заданный.
     *
     * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
     * @version 3.0
     * @since 1.0
     */
    REMOVE_GREATER {
        private int updateRow;

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            msgToClient = "Objects removed";
            try {
                connection.setAutoCommit(false);
                Statement statement = connection.createStatement();

                int age = newData.values().iterator().next().getAge();

                updateRow = statement.executeUpdate("DELETE FROM people WHERE age > " + age);

                connection.commit();
            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            }
            return updateRow;
        }
    },


    /**
     * Команда remove_all.
     * Удалят из коллекции все элементы, эквивалентные заданному.
     *
     * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
     * @version 3.0
     * @since 1.0
     */
    REMOVE_ALL {
        private int updateRow;

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            msgToClient = "Objects removed";
            try {
                connection.setAutoCommit(false);
                Statement statement = connection.createStatement();

                int age = newData.values().iterator().next().getAge();

                updateRow = statement.executeUpdate("DELETE FROM people WHERE age = " + age);

                connection.commit();
            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            }
            return updateRow;
        }
    },


    /**
     * Команда remove_lower.
     * Удаляет из коллекции все элементы, меньшие, чем заданный.
     *
     * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
     * @version 3.0
     * @since 1.0
     */
    REMOVE_LOWER_OBJECT {
        private int updateRow;

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            msgToClient = "Objects removed";
            try {
                connection.setAutoCommit(false);
                Statement statement = connection.createStatement();

                int age = newData.values().iterator().next().getAge();

                updateRow = statement.executeUpdate("DELETE FROM people WHERE age < " + age);

                connection.commit();
            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            }
            return updateRow;
        }
    },


    /**
     * Команда: add_if_max.
     * Добавляет новый элемент в коллекцию, если его значение превышает значение наибольшего элемента этой коллекции.
     *
     * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
     * @version 3.0
     */
    ADD_IF_MAX {
        private int updateRow;
        private boolean isMax = true;

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            try {
                Statement statement = connection.createStatement();

                ResultSet resultSet = statement.executeQuery("SELECT AGE " +
                        "FROM people " +
                        "ORDER BY AGE;");

                Iterator<Man> iterator = newData.values().iterator();
                int age = iterator.next().getAge();

                if (resultSet.getFetchSize() != 0) {
                    isMax = resultSet.getInt(1) > age;
                }

                this.updateRow = isMax ?
                        insertPeopleQueryExecute(connection, newData) :
                        0;

                msgToClient = isMax ?
                        "Object added" :
                        "Object is not max";

            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            }
            return updateRow;
        }
    },


    /**
     * Команда add_if_min.
     * Добавляет новый элемент в коллекцию, если его значение меньше, чем у наименьшего элемента этой коллекции.
     *
     * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
     * @version 2.0
     */
    ADD_IF_MIN {
        private int updateRow;
        private boolean isMin = true;

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            try {
                Statement statement = connection.createStatement();

                ResultSet resultSet = statement.executeQuery("SELECT AGE " +
                        "FROM people " +
                    "ORDER BY AGE;");

                Iterator<Man> iterator = newData.values().iterator();
                int age = iterator.next().getAge();

                if (resultSet.getFetchSize() != 0) {
                    isMin = resultSet.getInt(1) > age;
                }

                this.updateRow = isMin ?
                        insertPeopleQueryExecute(connection, newData) :
                        0;

                msgToClient = isMin ?
                        "Object added" :
                        "Object is not min";

            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            }
            return updateRow;
        }
    },


    /**
     * Команда import.
     * добавляет в коллекцию все данные из файла.
     *
     * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
     * @version 3.0
     */
    IMPORT_ALL_FROM_FILE {
        int updateRow;

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            msgToClient = "Object added";
            try {
                removeFromNewDataDuplicate(connection, newData);
                updateRow = insertPeopleQueryExecute(connection, newData);
            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            } catch (NumberFormatException e) {
                msgToClient = "Key is not correct";
            }
            return updateRow;
        }

        private void removeFromNewDataDuplicate(Connection connection, Map<String, Man> newData) throws SQLException {
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("SELECT ID FROM people");
            while (resultSet.next()) {
                if (newData.containsKey(String.valueOf(resultSet.getInt(1)))) {
                    newData.remove(String.valueOf(resultSet.getInt(1)));
                }
            }

        }
    },


    /**
     * Команда insert.
     * Добавляет новый элемент с заданным ключом.
     *
     * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
     * @version 3.0
     */
    INSERT_NEW_OBJECT {
        private boolean isInDB;
        private int updateRow;

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            try {
                Statement statement = connection.createStatement();
            int key = Integer.parseInt(newData.entrySet().iterator().next().getKey());

            isInDB = statement.executeQuery("SELECT ID " +
                    "FROM people " +
                    "WHERE ID = " + key).next();


            msgToClient = !isInDB ? "Object added" : "Object already in DB";

            updateRow = !isInDB ?
                    insertNewRowQuery(connection, newData) :
                    0;

        } catch (SQLException e) {
            msgToClient = "Could not connect to DB";
        } catch (NumberFormatException e) {
            msgToClient = "Key is not correct";
        }
        return updateRow;
    }
},


/**
 * Команда clear.
 * Очищает коллекцию.
 *
 * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
 * @version 3.0
 * @since 1.0
 */
CLEAR {
    @Override
    public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
        try {
            Statement statement = connection.createStatement();
            msgToClient = "Database cleared";
            return statement.executeUpdate("DELETE FROM people;");
        } catch (SQLException e) {
            msgToClient = "Could not connect to DB";
        }
        return 0;
    }
},


/**
 * Команда load.
 * Загружает дефолтные объекты типа {@link Storage} данные в коллекцию.
 *
 * @param peopleTree Ожидается TreeView<Container> для изменения содержимого
 * @version 3.0
 */
LOAD {
    private int updateRow;

    @Override
    public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
        try {
            connection.setAutoCommit(false);

            CLEAR.execute(connection, family, newData);

            updateRow = INSERT_NEW_OBJECT.execute(connection, family, newData);

            connection.commit();
            msgToClient = "Default data was loaded";
        } catch (SQLException e) {
            msgToClient = "Could not connect to DB";
        }
        return updateRow;
    }
},


READ {

},
    UPDATE {
        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            try {
                PreparedStatement statement = connection.prepareStatement(UPDATE_PEOPLE_NAME_QUERY);


                statement.setString(1, newData.values().iterator().next().getName());
                statement.setInt(2, Integer.parseInt(newData.keySet().iterator().next()));

                statement.executeUpdate();

            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            } catch (NumberFormatException e) {
                msgToClient = "Key is not correct";
            }
            return 0;
        }
    },


    REGISTER {
        private String createTable = "CREATE TABLE USERS(" +
                "NAME TEXT NOT NULL," +
                "PASSWORD TEXT NOT NULL," +
                "MAIL TEXT ," +
                "FULL_VERSION BOOLEAN NOT NULL," +
                "PRIMARY KEY(NAME, PASSWORD)" +
                ");";

        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            msgToClient = "This user already exist";
            try {

                try (Statement createTableStatement = connection.createStatement()) {
                    createTableStatement.executeUpdate(createTable);
                } catch (SQLException e) {
//                    System.out.println(e.getMessage());
                }

                Map.Entry<String, Man> user = newData.entrySet().iterator().next();

                String searchQuery = "SELECT count(*) FROM users WHERE name = ? AND mail = ?;";
                PreparedStatement searchStatement = connection.prepareStatement(searchQuery);
                searchStatement.setString(1, user.getValue().getName());
                searchStatement.setString(2, user.getKey());
                ResultSet resultSet = searchStatement.executeQuery();
                resultSet.next();
                if (resultSet.getInt(1) > 0) {
                    return 0;
                }
                searchStatement.close();


                String insertQuery = "INSERT INTO users VALUES (?,?,?);";
                PreparedStatement preparedStatement = connection.prepareStatement(insertQuery);
                preparedStatement.setString(1, user.getValue().getName());
                preparedStatement.setString(2, user.getKey());
                preparedStatement.setBoolean(3, user.getValue().getAge() == 2);
                preparedStatement.executeUpdate();

                msgToClient = "User added";
                return 1;
            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            }
            return 0;
        }
    },

    /**
     * age 1 - limited version
     * age 2 - full version
     * <p>
```

```
 * key - mail
 * name - username
 */
REGISTER_FULL {
    @Override
    public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
        msgToClient = "This user already exist";
        try {
            Map.Entry<String, Man> user = newData.entrySet().iterator().next();

            String searchQuery = "SELECT count(*) FROM users WHERE name = ? AND password = ?;";
            PreparedStatement searchStatement = connection.prepareStatement(searchQuery);
            searchStatement.setString(1, user.getValue().getName());
            searchStatement.setString(2, user.getKey());
            ResultSet resultSet = searchStatement.executeQuery();
            resultSet.next();
            if (resultSet.getInt(1) > 0) {
                return 0;
            }
            searchStatement.close();

            String password = generatePassword(connection, user);

            String insertQuery = "INSERT INTO users VALUES (?,?,?,?);";
            PreparedStatement preparedStatement = connection.prepareStatement(insertQuery);
            preparedStatement.setString(1, user.getValue().getName());
            preparedStatement.setString(2, password);
            preparedStatement.setBoolean(3, user.getValue().getAge() == 2);
            preparedStatement.setString(4, user.getKey());
            preparedStatement.executeUpdate();

            sendMessage(user.getKey(), user.getValue().getName(), password);

            msgToClient = "Password sent to e-mail " + user.getKey();
            return 1;
        } catch (SQLException e) {
            msgToClient = "Could not connect to DB";
        }
        return 0;
    }

    private void sendMessage(String username, String name, String password) {
        Properties props = new Properties();
        try (InputStream inputStream = Button.class.getResourceAsStream("/properties/mail.properties")) {
            props.load(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            Session session = Session.getDefaultInstance(props,
                    new Authenticator() {
                        protected PasswordAuthentication getPasswordAuthentication() {
                            return new PasswordAuthentication(props.getProperty("username"), props.getProperty("password"));
                        }
                    });

            // -- Create a new message --
            Message msg = new MimeMessage(session);

            // -- Set the FROM and TO fields --
            msg.setFrom(new InternetAddress(props.getProperty("username")));
            msg.setRecipients(Message.RecipientType.TO,
                    InternetAddress.parse(username, false));
            msg.setSubject("Collection");

            msg.setText("Username - " + name + "\n" + "Password - " + password);
            msg.setSentDate(new java.util.Date());
            Transport.send(msg);
            System.out.println("Message sent.");
        } catch (MessagingException e) {
            msgToClient = "E-mail isn't correct";
        }
    }

    private String generatePassword(Connection connection, Map.Entry<String, Man> user) throws SQLException {
        ResultSet resultSet;
        StringBuilder randString;
```

```
            PreparedStatement preparedStatement;

            int count ;
            do{
                count =(int) (Math.random() * 30);
            }while (count<7);

            do {
                String symbols = "qwertyuiopasdfghjklzxcvbnm1234567890";
                randString = new StringBuilder();
                for (int i = 0; i < count; i++) {
                    randString.append(symbols.charAt((int) (Math.random() * symbols.length())));
                }

                String insertQuery = "SELECT count(*) FROM users WHERE name = ? AND password = ? AND full_version = ? AND mail
= ?;";
                preparedStatement = connection.prepareStatement(insertQuery);
                preparedStatement.setString(1, user.getValue().getName());
                preparedStatement.setString(2, user.getKey());
                preparedStatement.setBoolean(3, user.getValue().getAge() == 2);
                preparedStatement.setString(4, String.valueOf(randString));
                resultSet = preparedStatement.executeQuery();
                resultSet.next();
            } while (resultSet.getInt(1) > 0);
            return String.valueOf(randString);
        }
    },


    /**
     * msgToClient : true - fullVersion; false - limitedVersion
     * return 1 if this user exist
     * 0 if this user doesn't exist
     */
    LOGIN {
        @Override
        public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
            msgToClient = "User not found";
            try {
                Statement searchStatement = connection.createStatement();


                Map.Entry<String, Man> user = newData.entrySet().iterator().next();

                String searchQuery = "SELECT full_version FROM users WHERE name = ? AND password = ?";
                PreparedStatement preparedStatement = connection.prepareStatement(searchQuery);
                preparedStatement.setString(1, user.getValue().getName());
                preparedStatement.setString(2, user.getKey());
                ResultSet resultSet = preparedStatement.executeQuery();

//                resultSet.next();
                if (resultSet.next()) {
                    msgToClient = String.valueOf(resultSet.getBoolean(1));
                    return 1;
                }
                searchStatement.close();
                return 0;
            } catch (SQLException e) {
                msgToClient = "Could not connect to DB";
            }
            return 0;
        }
    };


    private static final String INSERT_PEOPLE_QUERY =
            "INSERT INTO PEOPLE(AGE, NAME) VALUES (?,?);";
    private static final String INSERT_NEW_ROW_QUERY =
            "INSERT INTO PEOPLE VALUES(?,?,?)";
    private static final String UPDATE_PEOPLE_NAME_QUERY =
            "UPDATE PEOPLE SET name = ? WHERE id = ?;";
    private static String msgToClient;

    public static String getMsgToClient() {
        return msgToClient;
    }

    public int execute(Connection connection, Map<String, Man> family, Map<String, Man> newData) {
        return -1;
    }


    public int insertPeopleQueryExecute(Connection connection, Map<String, Man> newData) throws SQLException {
        int updateRow = 0;

        connection.setAutoCommit(false);
        PreparedStatement preparedStatement = connection.prepareStatement(INSERT_PEOPLE_QUERY);
        for (Map.Entry<String, Man> entry : newData.entrySet()) {
            preparedStatement.setInt(1, entry.getValue().getAge());
            preparedStatement.setString(2, entry.getValue().getName());
            updateRow += preparedStatement.executeUpdate();
        }
        connection.commit();

        return updateRow;
    }

    public int insertNewRowQuery(Connection connection, Map<String, Man> newData) {
        int updateRow = 0;
        try {
            connection.setAutoCommit(false);
            PreparedStatement preparedStatement = connection.prepareStatement(INSERT_NEW_ROW_QUERY);

            for (Map.Entry<String, Man> entry : newData.entrySet()) {
                preparedStatement.setInt(1, Integer.parseInt(entry.getKey()));
                preparedStatement.setInt(2, entry.getValue().getAge());
                preparedStatement.setString(3, entry.getValue().getName());
                updateRow += preparedStatement.executeUpdate();
            }

            connection.commit();
        } catch (SQLException e) {
            if (e.getSQLState().equals("23514")) {
                msgToClient = "Age should be positive";
            }
        }
        return updateRow;
    }
}
```

class WorkWithDB

```
package connectDB;


import GUI.Button;
import old.school.Man;
import old.school.People;
import org.postgresql.ds.PGConnectionPoolDataSource;

import javax.sql.PooledConnection;
import java.io.*;
import java.sql.*;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Properties;

/**
 * Created by slavik on 30.04.17.
 */
public class WorkWithDB {

    private static final String FILE_NAME_DB_PROPERTIES = "DataBase.properties";
    private ByteArrayOutputStream oldByteData;
    private ByteArrayOutputStream newByteData;
    private Map<String, Man> family;
    private Map<String, Man> newData;
    private Button button;


    WorkWithDB() {
    }

    public WorkWithDB(ByteArrayOutputStream oldByteData, ByteArrayOutputStream newByteData, Button button) {
        this.oldByteData = oldByteData;
        this.newByteData = newByteData;
        this.button = button;
    }


    public MessageToClient executeCommand() {
        try {

            Connection connection = getConnection();
            Statement statement = connection.createStatement();

            initTable(statement);

            deserializeInputData();

            MessageToClient messageToClient = new MessageToClient(checkOldData(statement), modifyDataInDB(connection),
getNewDataForClient(statement), Button.getMsgToClient());
            connection.close();
            return messageToClient;
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        return null;
    }

    private Map<String, Man> getNewDataForClient(Statement statement) {
        Map<String, Man> dataFromDB = new LinkedHashMap<>();
        try {
            ResultSet resultSet = statement.executeQuery("SELECT * FROM people");
            while (resultSet.next()) {
                dataFromDB.put((String.valueOf(resultSet.getInt(1))), new People(resultSet.getInt(2), resultSet.getString(3)));
            }
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        return dataFromDB;
    }

    private int modifyDataInDB(Connection connection) {
        return button.execute(connection, family, newData);
    }


    //true - data in DB and data from client is equals
    private boolean checkOldData(Statement statement) {
        try {
            ResultSet sizeResultSet = statement.executeQuery("SELECT count(*) FROM people;");
            sizeResultSet.next();
            int size = sizeResultSet.getInt(1);

            ResultSet resultSet = statement.executeQuery("SELECT * FROM people;");

            if (size != family.size()) {
                return false;
            }

            while (resultSet.next()) {
                if (!(family.containsKey(resultSet.getString(1))&&
                        family.get(resultSet.getString(1)).getName().equals(resultSet.getString(3))&&
                        family.get(resultSet.getString(1)).getAge()==resultSet.getInt(2))) {
                    return false;
                }
            }
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        return true;
    }

    private void initTable(Statement statement) {
        String createTable = "CREATE TABLE PEOPLE(\n" +
                "  ID SERIAL PRIMARY KEY,\n" +
                "  AGE INTEGER CONSTRAINT positive_age CHECK (AGE>=0) NOT NULL,\n" +
                "  NAME TEXT \n" +
                ");";
        try {
            statement.executeUpdate(createTable);
        } catch (SQLException e) {
//            System.out.println(e.getMessage());
        }
    }

    private void deserializeInputData() {
        ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(newByteData.toByteArray());
        try (ObjectInputStream objectInputStream = new ObjectInputStream(byteArrayInputStream)) {
            newData = (Map<String, Man>) objectInputStream.readObject();
        } catch (IOException | ClassNotFoundException e) {
            System.out.println(e.getMessage());
        }

        try (ObjectInputStream objectInputStream = new ObjectInputStream(new ByteArrayInputStream(oldByteData.toByteArray()))) {
            family = (Map<String, Man>) objectInputStream.readObject();
        } catch (IOException | ClassNotFoundException e) {
            System.out.println(e.getMessage());
        }

    }

    private Connection getConnection() throws SQLException {
        Properties dataBaseProperties = getProperties();
        PGConnectionPoolDataSource pgConnectionPoolDataSource = new PGConnectionPoolDataSource();
        pgConnectionPoolDataSource.setDatabaseName(dataBaseProperties.getProperty("jdbs.dbname"));
        pgConnectionPoolDataSource.setServerName(dataBaseProperties.getProperty("jdbs.servername"));
        PooledConnection pooledConnection =
pgConnectionPoolDataSource.getPooledConnection(dataBaseProperties.getProperty("jdbs.username"),
dataBaseProperties.getProperty("jdbs.password"));

        return pooledConnection.getConnection();
    }

    private Properties getProperties() {

        Properties platformProperties = new Properties();
        try (InputStream scanner = WorkWithDB.class.getResourceAsStream("/properties/" + FILE_NAME_DB_PROPERTIES)) {
            platformProperties.load(scanner);
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println("Could not connect");
        }


        Properties dataBaseProperties = new Properties();
        try (InputStream scanner = WorkWithDB.class.getResourceAsStream("/properties/" + platformProperties.getProperty("platform")))
{
            dataBaseProperties.load(scanner);
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println("Could not connect");
        }

        return dataBaseProperties;
    }

}
```

class MessageToClient

```
package connectDB;

import old.school.Man;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;
import java.util.Map;

/**
 * Created byteArrayOutputStream slavik on 02.05.17.
 */
```

```
public class MessageToClient {
    private boolean clientCollectionState;
    private int modifiedRow;
    private Map<String, Man> dataToClient;
    private ByteBuffer outputData;
    private String msgToClient;
    private ByteArrayOutputStream byteArrayOutputStream;
    private ObjectOutputStream objectOutputStream;

    MessageToClient(boolean clientCollectionState, int modifiedRow, Map<String, Man> dataToClient, String msgToClient) {
        this.clientCollectionState = clientCollectionState;
        this.modifiedRow = modifiedRow;
        this.dataToClient = dataToClient;
        this.msgToClient = msgToClient;
    }

    //newData, NEW, modifiedRow, STATE, clientConnectionState, MSG, msgToClient, END
    public void sendData(DatagramChannel serverSocket, SocketAddress socketAddress) {
        try  {
            sendMap(serverSocket, socketAddress);
            sendServiceInformation(serverSocket, socketAddress, "NEW");
            sendModifiedRow(serverSocket, socketAddress);
            sendServiceInformation(serverSocket, socketAddress, "STATE");
            sendClientCollectionState(serverSocket, socketAddress);
            sendServiceInformation(serverSocket, socketAddress, "MSG");
            sendMsgToClient(serverSocket,socketAddress);
            sendServiceInformation(serverSocket, socketAddress, "END");

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

    private void sendMsgToClient(DatagramChannel serverSocket, SocketAddress socketAddress) throws IOException {
        byteArrayOutputStream = new ByteArrayOutputStream();
        objectOutputStream = new ObjectOutputStream(byteArrayOutputStream);
        objectOutputStream.writeObject(msgToClient);
        objectOutputStream.flush();
        outputData = ByteBuffer.wrap(byteArrayOutputStream.toByteArray());
        serverSocket.send(outputData,socketAddress);
    }

    private void sendServiceInformation(DatagramChannel serverSocket, SocketAddress socketAddress, String serviceInformation)
throws IOException {
        outputData = ByteBuffer.wrap(serviceInformation.getBytes());
        serverSocket.send(outputData, socketAddress);
    }

    //client data state
    private void sendClientCollectionState(DatagramChannel serverSocket, SocketAddress socketAddress) throws IOException {
        byteArrayOutputStream = new ByteArrayOutputStream();
        objectOutputStream = new ObjectOutputStream(byteArrayOutputStream);
        objectOutputStream.writeBoolean(clientCollectionState);
        objectOutputStream.flush();
        outputData = ByteBuffer.wrap(byteArrayOutputStream.toByteArray());
        serverSocket.send(outputData, socketAddress);
    }

    //how much was modified row
    private void sendModifiedRow(DatagramChannel serverSocket, SocketAddress socketAddress) throws IOException {
        byteArrayOutputStream = new ByteArrayOutputStream();
        objectOutputStream = new ObjectOutputStream(byteArrayOutputStream);
        objectOutputStream.writeInt(modifiedRow);
        objectOutputStream.flush();
        outputData = ByteBuffer.wrap(byteArrayOutputStream.toByteArray());
        serverSocket.send(outputData, socketAddress);
    }

    //new data
    private void sendMap(DatagramChannel serverSocket, SocketAddress socketAddress) throws IOException {
        byteArrayOutputStream = new ByteArrayOutputStream();
        this.objectOutputStream = new ObjectOutputStream(byteArrayOutputStream);
        this.objectOutputStream.writeObject(dataToClient);
        objectOutputStream.flush();
        outputData = ByteBuffer.wrap(byteArrayOutputStream.toByteArray());
        serverSocket.send(outputData, socketAddress);
    }
}
```