Homework 2

Violations of SOLID principles in code (my code)

# Single Responsibility and Open for extension, closed for modification violation

Code:

**public** **class** Builder<T> {

**private** Class<T> type;

**public** Builder(Class<T> type) {

**this**.type = type;

}

**public** PreparedStatement findByIdStatement(Connection connection, **int** id, String field) **throws** SQLException {

StringBuilder findById = **new** StringBuilder("SELECT \* FROM ");

findById.append(type.getSimpleName());

findById.append(" where ");

findById.append(field);

findById.append("=?");

PreparedStatement statement = connection.prepareStatement(findById.toString());

statement.setInt(1, id);

**return** statement;

}

**public** PreparedStatement findByNameStatement(Connection connection, String name, String field) **throws** SQLException {

StringBuilder findById = **new** StringBuilder("SELECT \* FROM ");

findById.append(type.getSimpleName());

findById.append(" where ");

findById.append(field);

findById.append("=?");

PreparedStatement statement = connection.prepareStatement(findById.toString());

statement.setString(1, name);

**return** statement;

}

**public** PreparedStatement updateByIdStatement(Connection connection, **int** id, Object[] values) **throws** SQLException {

StringBuilder update = **new** StringBuilder(" UPDATE ");

Field[] fs = type.getDeclaredFields();

update.append(type.getSimpleName());

update.append(" Set ");

**for** (**int** i = 0; i < fs.length; i++) {

**if** (!fs[i].getName().contains("id")) {

**if** (i != fs.length - 1)

update.append(fs[i].getName() + "=?, ");

**else**

update.append(fs[i].getName() + "=? ");

}

}

**if** (Perfume.**class**.isAssignableFrom(type))

update.append(" where PerfumeId=?");

**else**

update.append(" where Id=?");

PreparedStatement statement = connection.prepareStatement(update.toString());

**for** (**int** i = 1; i <= values.length; i++) {

**if** (values[i - 1] **instanceof** String)

statement.setString(i, (String) values[i - 1]);

**else** **if** (values[i - 1] **instanceof** Integer)

statement.setInt(i, (**int**) values[i - 1]);

**else**

statement.setLong(i, (**long**) values[i - 1]);

System.***out***.println(statement.toString());

}

statement.setInt(values.length + 1, id);

**return** statement;

}

**public** PreparedStatement findAllStatement(Connection connection, String field) **throws** SQLException {

StringBuilder findById = **new** StringBuilder("SELECT \* FROM ");

findById.append(type.getSimpleName());

findById.append(" order by ");

findById.append(field);

PreparedStatement statement = connection.prepareStatement(findById.toString());

**return** statement;

}

**public** PreparedStatement insertManytoMany(Connection connection, Object e, **int** id)

**throws** SQLException, IllegalArgumentException, IllegalAccessException {

StringBuilder insert = **new** StringBuilder("insert into ");

Field[] fs = type.getDeclaredFields();

**for** (Field f : fs)

f.setAccessible(**true**);

insert.append(e.getClass().getSimpleName() + " (");

**for** (**int** i = 0; i < fs.length; i++) {

**if** (i == fs.length - 1)

insert.append(fs[i].getName() + ")");

**else**

insert.append(fs[i].getName() + ", ");

}

insert.append("Values(");

**for** (**int** i = 0; i < fs.length; i++) {

**if** (i == fs.length - 1)

insert.append("?)");

**else**

insert.append("?,");

}

PreparedStatement statement = connection.prepareStatement(insert.toString());

statement.setInt(1, id);

**for** (**int** i = 1; i < fs.length; i++) {

**if** (fs[i].getType().isAssignableFrom(String.**class**))

statement.setString(i + 1, (String) fs[i].get(e));

**else** **if** (fs[i].getType().isAssignableFrom(Integer.**class**))

statement.setInt(i + 1, fs[i].getInt(e));

**else**

statement.setLong(i + 1, fs[i].getLong(e));

}

System.***out***.println(statement);

**return** statement;

}

**public** PreparedStatement insertStatement(Connection connection, Object e)

**throws** SQLException, IllegalArgumentException, IllegalAccessException {

StringBuilder insert = **new** StringBuilder("insert into ");

Field[] fs = type.getDeclaredFields();

insert.append(type.getSimpleName() + "(");

**for** (**int** i = 0; i < fs.length; i++) {

**if** (i == 0)

**continue**; // pass over objects with id field

**else** **if** (i == fs.length - 1)

insert.append(fs[i].getName() + " ");

**else**

insert.append(fs[i].getName() + ", ");

}

insert.append(") Values (?");

**for** (**int** i = 2; i < fs.length; i++)

insert.append(",?");

insert.append(")");

System.***out***.println(insert.toString());

PreparedStatement statement = connection.prepareStatement(insert.toString());

statement = setStatement(statement, e);

**return** statement;

}

**public** PreparedStatement insertChildStatement(Connection connection, Object e, **int** id)

**throws** SQLException, IllegalArgumentException, IllegalAccessException {

StringBuilder insert = **new** StringBuilder("insert into ");

Field[] fs = type.getDeclaredFields();

**for** (Field f : fs)

f.setAccessible(**true**);

**if** (e.getClass().equals(Perfume.**class**))

insert.append(type.getSimpleName() + "(Perfumeid,");

**else**

insert.append(type.getSimpleName() + "(id,");

**for** (**int** i = 0; i < fs.length; i++) {

**if** (i == fs.length - 1)

insert.append(fs[i].getName() + " ");

**else** **if** (i > 0)

insert.append(fs[i].getName() + ", ");

**else**

insert.append(fs[i].getName() + ", ");

}

insert.append(") Values (?,"); // for id

**for** (**int** i = 0; i < fs.length; i++) {

**if** (i == fs.length - 1)

insert.append("?)");

**else** **if** (i > 0)

insert.append(",?");

**else**

insert.append("?,");

}

PreparedStatement statement = connection.prepareStatement(insert.toString());

statement.setInt(1, id);

**for** (**int** i = 1; i <= fs.length; i++) {

**if** (fs[i - 1].getType().isAssignableFrom(String.**class**))

statement.setString(i + 1, (String) fs[i - 1].get(e));

**else** **if** (fs[i - 1].getType().isAssignableFrom(Integer.**class**))

statement.setInt(i + 1, fs[i - 1].getInt(e));

**else**

statement.setLong(i + 1, fs[i - 1].getLong(e));

System.***out***.println("child " + statement);

}

**return** statement;

}

**public** PreparedStatement deleteStatement(Connection connection, **int** id,String field) **throws** SQLException{

StringBuilder delete = **new** StringBuilder("delete from ");

delete.append(type.getSimpleName());

delete.append(" where ");

delete.append(field);

delete.append("=?");

PreparedStatement statement = connection.prepareStatement(delete.toString());

statement.setInt(1, id);

**return** statement;

}

**public** PreparedStatement setStatement(PreparedStatement statement, Object e)

**throws** IllegalArgumentException, IllegalAccessException, SQLException {

Field[] fs = e.getClass().getDeclaredFields();

**for** (Field f : fs)

f.setAccessible(**true**);

**if** (e.getClass().getSuperclass().equals(Product.**class**)) {

**for** (**int** i = 0; i < fs.length; i++) {

**if** (fs[i].getType().isAssignableFrom(String.**class**))

statement.setString(i + 1, (String) fs[i].get(e));

**else** **if** (fs[i].getType().isAssignableFrom(Integer.**class**))

statement.setInt(i + 1, fs[i].getInt(e));

**else**

statement.setLong(i + 1, fs[i].getLong(e));

}

} **else** {

**for** (**int** i = 1; i < fs.length; i++) {

**if** (fs[i].getType().isAssignableFrom(String.**class**))

statement.setString(i, (String) fs[i].get(e));

**else** **if** (fs[i].getType().isAssignableFrom(Integer.**class**))

statement.setInt(i, fs[i].getInt(e));

**else**

statement.setLong(i, fs[i].getLong(e));

}

}

**return** statement;

}

**public** PreparedStatement setDisplayedRows(Connection connection) **throws** SecurityException, SQLException {

PreparedStatement statement = **null**;

**if** (type.equals(Perfume.**class**)) {

statement = connection.prepareStatement(

"SELECT product.name,perfume.house , perfume.capacity, product.price, product.stock, product.id from product join perfume on product.id = perfume.perfumeid");

} **else** **if** (type.equals(Book.**class**)) {

statement = connection.prepareStatement(

"SELECT product.name,book.author, product.price, product.stock,product.id from product join book on product.id = book.id");

} **else**

statement = findAllStatement(connection, type.getDeclaredFields()[0].getName());

**return** statement;

}

The above class represents a nice example of managing to violate not only 1 SOLID principle. Its main purpose was to serve as a class to dynamically build SQL query statements for database access.

I believe it violates the O principle because it offers no flexibility for extension and covering other type of queries/other type of databases which use a different query language. Ideally, an interface with methods for all the desired queries should have been implemented, allowing different use cases and technologies to be covered.

Secondly, I believe it violates the S principle because it covers a lot of different type of statements, which differ in their purpose. Going accordingly to the previous idea of using interfaces, I believe that splitting a large interface in smaller ones such as UpdateStatementInterface, DeleteStatementInterface and so on would have increased the cohesion of the implementing classes. Consequently, this big class would have been split in 3+ smaller classes, easier to change and understand.

# Interface Segregation

Code

**public** **class** MenuController {

**private** MenuView menuView;

**private** Bank bank;

**public** MenuController(MenuView menuView, Bank bank){

**this**.menuView=menuView;

**this**.bank= bank;

**this**.menuView.addClientOpsListener(**new** ClientOpsListener());

**this**.menuView.addManagerOpsListener(**new** ManagerOpsListener());

//this.menuView.addExitListener(new WindowExitListener() );

}

**private** **class** ClientOpsListener **implements** ActionListener{

**private** ClientView db;

@Override

**public** **void** actionPerformed(ActionEvent arg0) {

ClientView cv= **new** ClientView();

**this**.openDatabase(cv);

ClientController cl = **new** ClientController(cv,bank);

db.getFrame().setVisible(**true**);

}

**public** **void** openDatabase(ClientView db) {

**this**.db = db;

}

}

**private** **class** ManagerOpsListener **implements** ActionListener{

**private** ManagerView db;

@Override

**public** **void** actionPerformed(ActionEvent arg0) {

ManagerView mv= **new** ManagerView();

**this**.openDatabase(mv);

ManagerController mc = **new** ManagerController(bank,mv);

db.getFrame().setVisible(**true**);

}

**public** **void** openDatabase(ManagerView db) {

**this**.db = db;

}

}

}

The above example represents the GUI for a simple Banking app. As it can be seen, it offers together for a user the possibility of accessing either Manager operations or Client operations, without any security measures. Ideally, we should split the interface in 2: client and manager. Then, we should offer the corresponding one to the user, after identifying its role beforehand.