Warehouse Manager

1. Objectives

Asdf

2. Problem Analysis

This chapter aims to describe the requirements of the problem as well as how the end-users will interact with the application, together with any assumptions made about the input of such end-users.

2.1. Functional Requirements

The completed application must provide the following functionality:

- Provide an interactive window for operations on users, items and orders
- Support the following operations: creation, reading, updating (editing) and deletion - the CRUD operations.
- Store necessary information about users, items and orders in a relational database.
- Use reflection for both database operations and display operations, so that it remains flexible to later expansion.
- Validate new order requests and display a message in case there is too little remaining stock to complete the order.
- At the user's request, generate a receipt for a given order in .txt format.

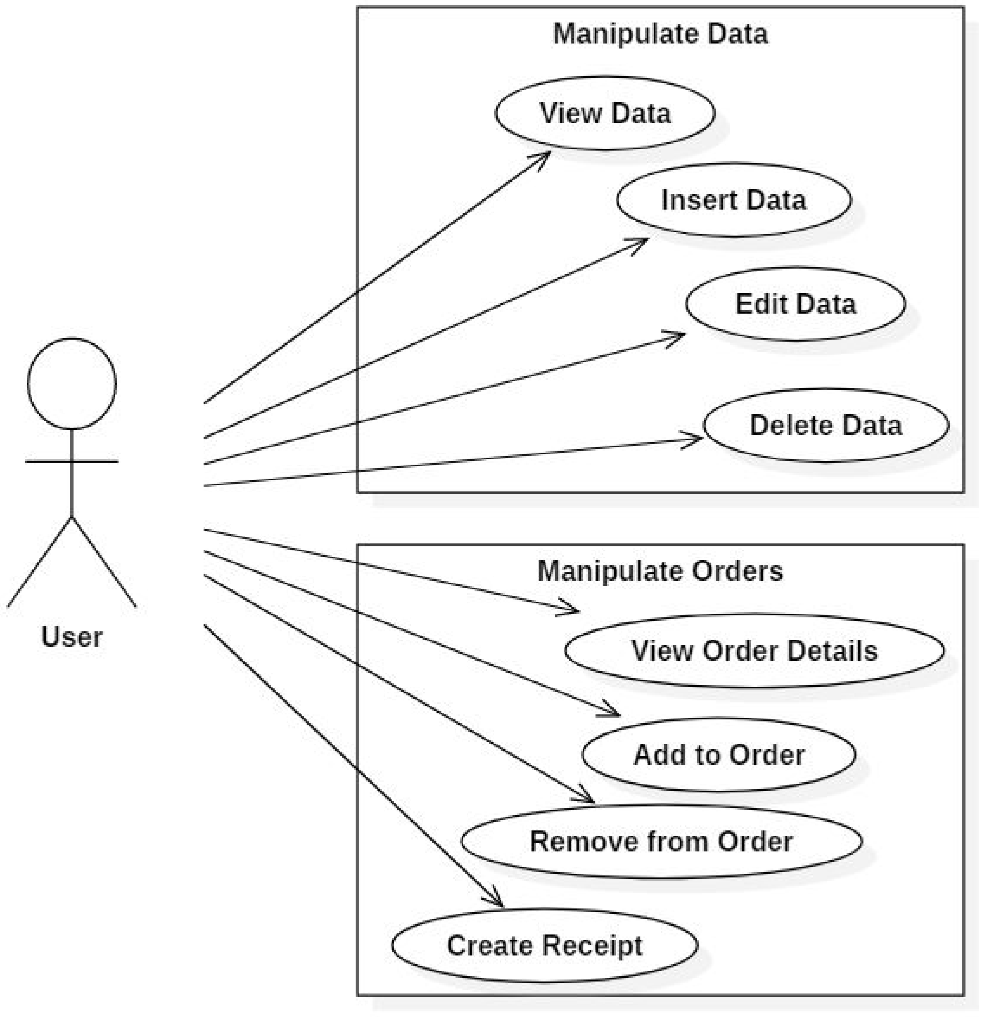
2.2. Assumptions

For an easier implementation of the application, we assume that the user will input correct data into all of the required fields or, in the case of errors, will correct the information before leaving the application. (That is, we assume that the data in the database will be valid once the user has finished operating.)

We also assume that the user knows how to operate the program or otherwise does need any instructions directly in the application.

2.3. Use Cases

The following diagram shows the possible use-cases for this application:



Use case: add an entry to an order (Add to Order)
Actors: end user
Preconditions:

- The database contains at least one item and one client
- There is at least one order associated to a client

Main success scenario:

1. The user selects the order to add an entry to
2. The program displays a detailed view of the order
3. The user selects an item to add to the order
4. The user chooses an amount of that item to be added to the order
5. The program validates that the item's stock is sufficient
6. The program subtracts the amount from the item's stock
7. The program adds an entry to the chosen order specifying the item and the amount
8. The program updates the database and the display with the changes

Alternative sequences:

A. Insufficient stock
    5. The program checks the stock and notices it is insufficient
    6. The program displays an "under stock" message
    7. The use case returns to step 2.

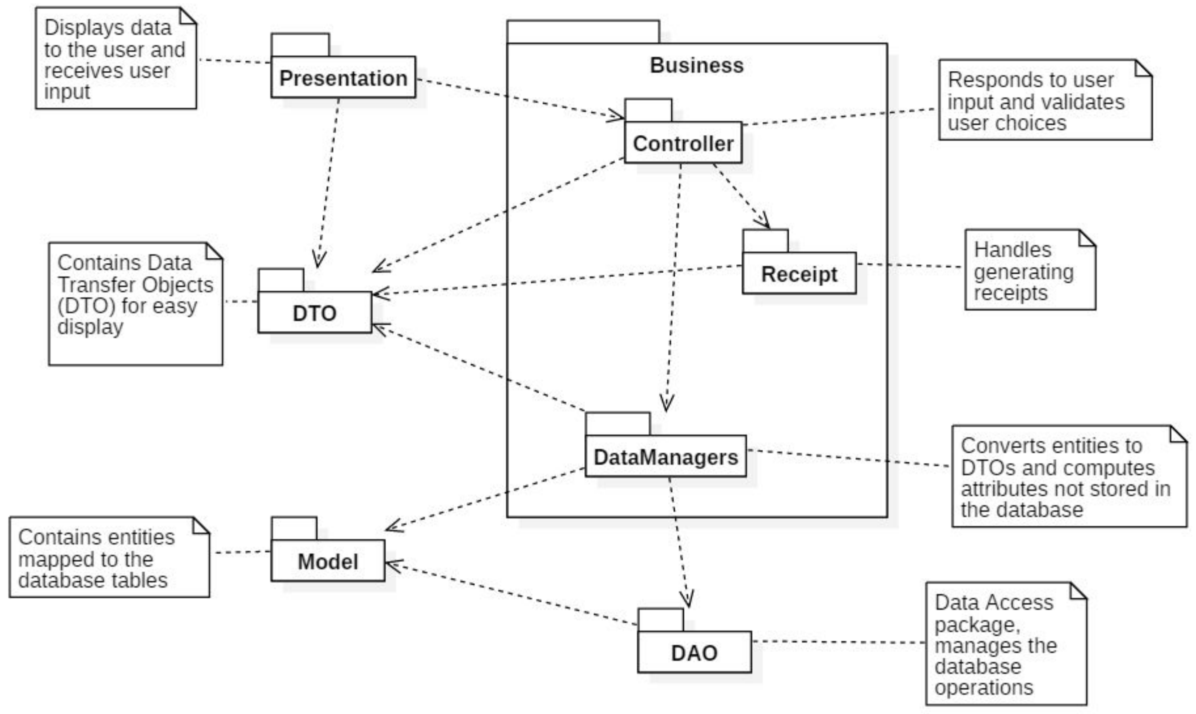
B. User cancellation
    4. The user decides to cancel the operation
    5. The use case returns to step 2.

3. Design

This chapter deals with how the the problem is approached from an object-oriented perspective, how it will be split into packages, what interfaces and data structures will be used and the algorithms that will be employed.

3.1. Package Distribution

The application will be constructed employing the Layers architectural pattern. Below is the intended package diagram for the application, together with package dependencies:


Displays data to the user and receives user input
Presentation
Business
Controller
Responds to user input and validates user choices
Receipt
Handles generating receipts
Contains Data Transfer Objects (DTO) for easy display
DTO
DataManagers
Converts entities to DTOs and computes attributes not stored in the database
Contains entities mapped to the database tables
Model
DAO
Data Access package, manages the database operations

We plan, if feasible within the time budget of the project, to obtain a dependency inversion between the Presentation and Controller packages, as in the image. We plan on achieving this by declaring an interface in the Controller package that the Presentation must follow, so that although the controller owns a view, the dependency goes the other way around.

3.2. Class Design

In this subchapter we describe what kind of classes we wish to implement in each package and give reasons for our choices. Although a class diagram containing all the classes of the project would be infeasible, we will include smaller class diagrams where appropriate

3.2.1. Entities

There will be one entity class for each table in the database and the fields of the entity will correspond to the columns in the database. This is necessary to facilitate using reflection for database access.

We expect there to be one entity class for each table in the database, which means four in total: one for items, one for clients, one for orders and one for the many-to-many association between orders and items.

3.2.2. DTOs

Some features that we'd like to show in the user interface, such as the total cost of an order, would be cumbersome to store in the database and keep updated. Because of this, our entities do not contain these features. However, it would be useful to have a kind of object that contains such information, so that these objects can be easily displayed.

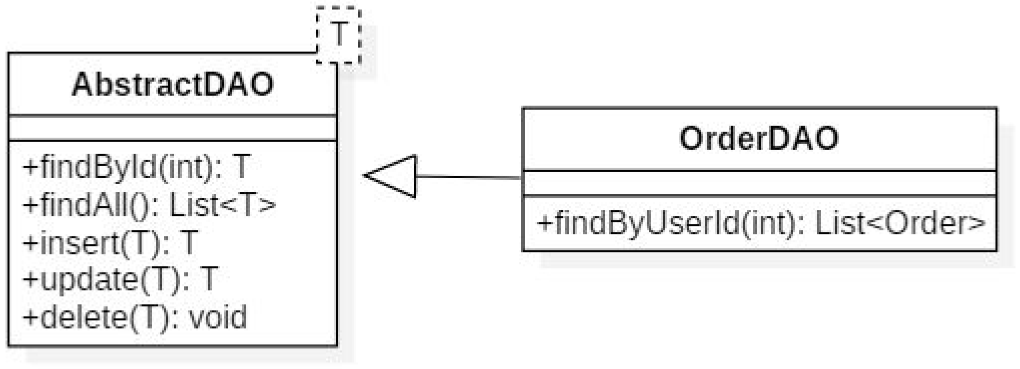
As a result, we implement data transfer objects that store such information. For example, the DTO for an order will contain, in addition to the normal data an order entity provides, a field storing the total cost of the items contained in the order. This field will be computed when the DTO is created.

3.2.3. DAOs

We will use data access objects (DAOs) to interact with the database, so that the exact implementation of our database is encapsulated from the rest of our program.

While each entity class will have its own DAO, much of the code is the same for any object (typical calls such as “select all”, “insert”, “delete”, “update”, “find by id”). Because of this, we will abstract typical calls into an abstract DAO class that uses reflection to handle any type of data, and will only implement specific methods (such as finding all the orders associated to a specific user) in the other DAOs.

Below is an example of how this system will work:



The other DAO classes will similarly be obtained by inheriting from AbstractDAO, where most of the implementation will live. This makes it easy to add new entities, since their DAOs only need to implement functions for specific queries.

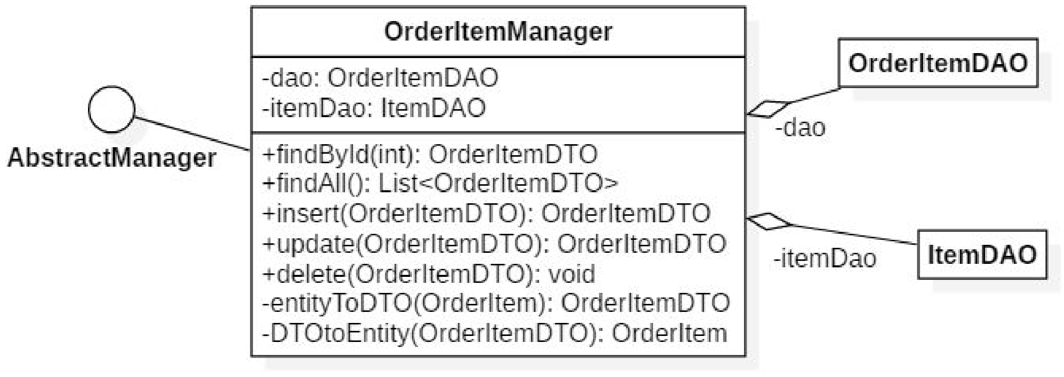
Finally, we will use a MySql database that we will access using JDBC, although it should be possible to change this at a later date by simply providing a different DAO implementation.

3.2.4. Data Managers

Data managers serve to convert our entities, which are not directly useful, into the more useful and easy to display DTOs. They also compute features such as the total price of an order which are not stored in the database.

Because of their functionality, the managers act as a wrapper around the DAO classes - indeed, every manager simply delegates most of its work to DAO. However, these tasks cannot be properly done in the DAO package, because doing so would require us to abandon the usage of reflection.

Below is an example of how a data manager might look:


OrderItemManager
-dao: OrderItemDAO
-itemDao: ItemDAO
+findById(int): OrderItemDTO
+findAll(): List<OrderItemDTO>
+insert(OrderItemDTO): OrderItemDTO
+update(OrderItemDTO): OrderItemDTO
+delete(OrderItemDTO): void
-entityToDTO(OrderItem): OrderItemDTO
-DTOtoEntity(OrderItemDTO): OrderItem
AbstractManager
OrderItemDAO
-dao
ItemDAO
-itemDao

Notice how a manager of OrderItems (the entities used for the many-to-many mapping Item - Order, which also store how many items of that kind are included in the order) needs a DAO for OrderItems and also a DAO for Items. This is because, in order to obtain information such as the subtotal for an OrderItem, we need to find out the cost of its associated item.

3.2.5. Receipt

It is likely that, given the simplicity of our table structure and how we obtain the receipts in .txt format (and never need to print, e-mail or store them otherwise), our Receipt package will only contain one class. This class, ReceiptWriter, will only need to provide two functions: compose the receipt from a list of DTOs (all the information needed, such as totals and subtotals, is already included in the DTOs, so the only thing that needs to be done is text formatting) and output it to a file.

In the future, the receipt-emitting functionality might grow more complex, so we have separated it into a different package early in development, although for now it would be acceptable for ReceiptWriter to simply reside in the Controller package.

3.2.6. Presentation

In this package we will employ reflection again, in order to create a versatile view that can display any kind of data and to which listeners can be attached so that the controller can respond to events.

Because we plan to have a generic view class, the importance of using DTOs is emphasised: if we had tried to use the table-mapped entities, it would have been impossible to display data such as totals and subtotals, which are not mapped to any database columns, unless we computed them separately - and doing so in the presentation would have invalidated using generics.

We expect to need to display the following:

- An interactive window showing the items
- An interactive window showing the clients
- An interactive window showing the orders

- An interactive window showing the OrderItem entries related to a currently inspected order (a detail view much like viewing one’s shopping cart)
- A main window that controls the others so that we can avoid unneeded clutter

We conclude that the first three can be achieved with a single generic view, but that the latter two need individual views, because their functionalities are different.

Moreover, we can still apply listeners (for instance, to detect when the user edits a table entry) by simply outsourcing the act of connecting listeners to the tables, buttons and anything else to the Controller.

3.2.7. Controller

The Controller package deals with connecting the database / DAO / data managers part of the application to the presentation / UI part. Because of this, it is the most complex part of the program and the hardest to predict in advance.

We predict that, because it needs to control at least three different types of views (and probably five views in total), it is likely that the control will need to be split across multiple classes.

On the other hand, changes in one view can affect the others - for instance, changing the price of an item cascades into the OrderItem and order views, since their attributes are affected by the item’s price. While the most elegant solution to this would be something along the lines of the Observer pattern, if complexity is sufficiently low, it might still be possible to have a monolithic controller.

Should a monolithic controller be the final result, it is possible that certain segments of code such as table listeners will be generalised and moved into separate classes.

As a conclusion, we are not yet certain how the Controller’s functionality will be implemented, since it is quite strongly affected by other implementation details. We will leave this topic to the next chapter, which will be tackled once the classes are already fleshed out.

4. Implementation

This chapter describes the classes and interfaces in more detail, covering the most important methods and aspects of each.

Due to constraints on the length of this documentation, we will only include the most relevant areas of code here, while attaching the entire code separately, where the Javadoc will serve as reading guide if needed.

Moreover, we will focus on the classes that are of higher importance (for example, we will skip entities entirely and provide only one example for classes that are similar to each other, such as data managers).

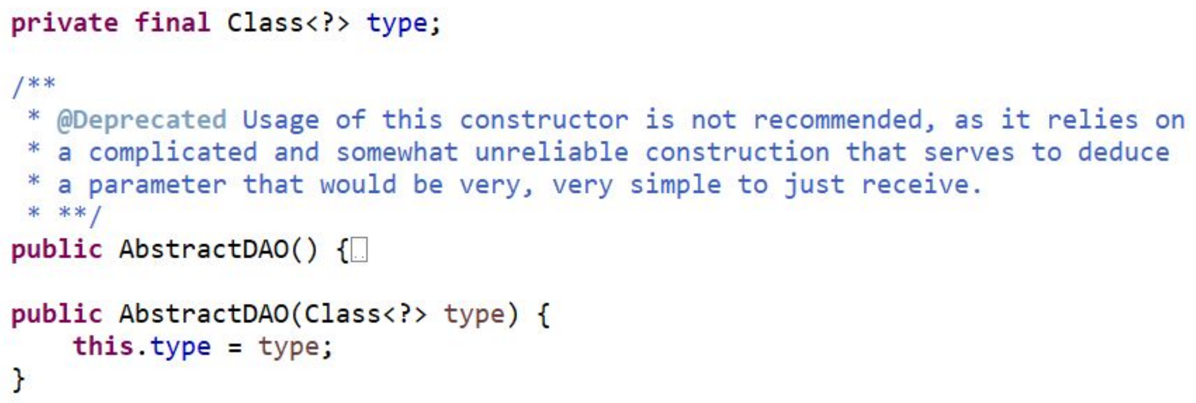
4.0. Entities

Before we begin analysing code, we will briefly highlight the final structure of the entities:

- Item represents items and stores an id, a name and a decimal price (stores two digits after the point, like real currencies)
- User represents the clients and stores an id and an e-mail address. It has been implemented so because the important information about warehouse users was the way to contact them.
- Order represents the orders and stores an id and the associated user’s id (the total price information is not stored, but instead computed)
- OrderItem represents the connection between Order and Item and stores an id, an order id, an item id and a numeric amount (the number of items of that kind associated to this particular order)

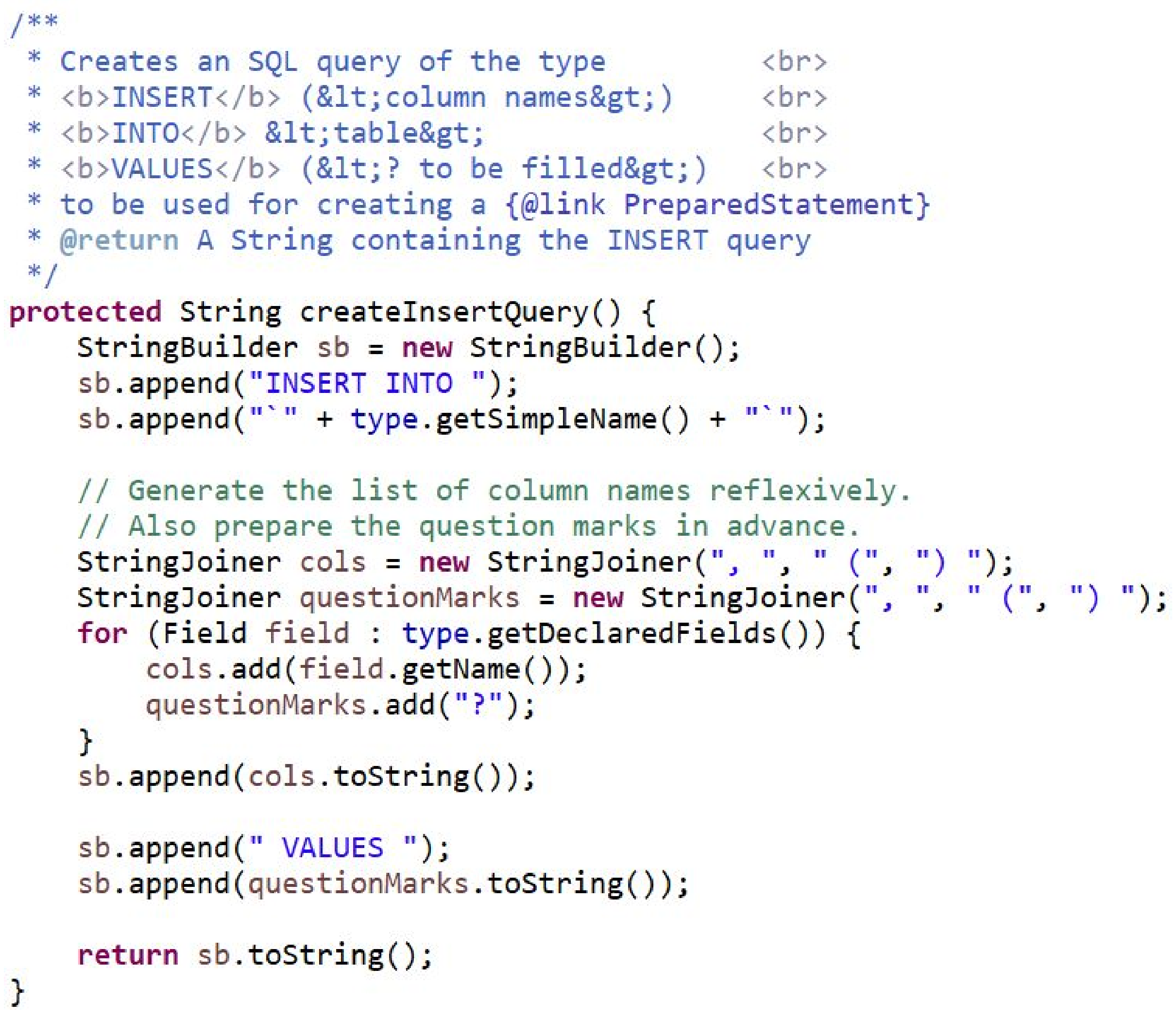
4.1. DAOs

Continuing what we have stated in chapter 3.2.3., we have implemented database access using a generic class AbstractDAO<T>, which is then extended for each entity. In the following we present a number of relevant code fragments from AbstractDAO.



```java
private final Class<?> type;

/**
 * @Deprecated Usage of this constructor is not recommended, as it relies on
 * a complicated and somewhat unreliable construction that serves to deduce
 * a parameter that would be very, very simple to just receive.
 * **/
public AbstractDAO() {□

public AbstractDAO(Class<?> type) {
    this.type = type;
}
```

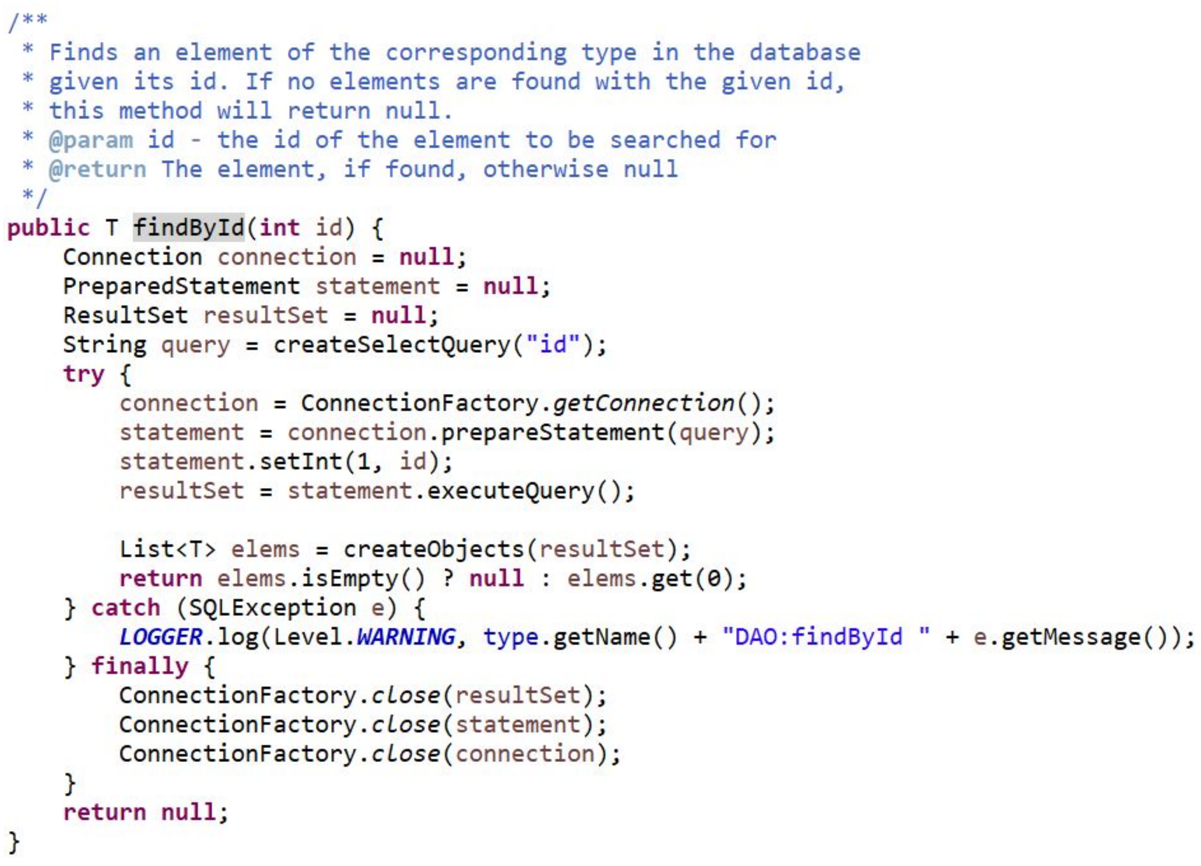
AbstractDAO stores an attribute that tells it the class of its generic type. It is useful to have such an attribute, as it eases all tasks regarding reflection. It is possible to use a Java artifice to obtain this value at runtime, but we believe it is far cleaner to simply have the one constructing a DAO pass us this attribute (we will see that classes that inherit this one also pass this attribute).



```java
/**
 * Creates an SQL query of the type          <br>
 * <b>INSERT</b> (&lt;column names&gt;)      <br>
 * <b>INTO</b> &lt;table&gt;                 <br>
 * <b>VALUES</b> (&lt;? to be filled&gt;)    <br>
 * to be used for creating a {@link PreparedStatement}
 * @return A String containing the INSERT query
 */
protected String createInsertQuery() {
    StringBuilder sb = new StringBuilder();
    sb.append("INSERT INTO ");
    sb.append("`" + type.getSimpleName() + "`");

    // Generate the list of column names reflexively.
    // Also prepare the question marks in advance.
    StringJoiner cols = new StringJoiner(", ", " (", ") ");
    StringJoiner questionMarks = new StringJoiner(", ", " (", ") ");
    for (Field field : type.getDeclaredFields()) {
        cols.add(field.getName());
        questionMarks.add("?");
    }
    sb.append(cols.toString());

    sb.append(" VALUES ");
    sb.append(questionMarks.toString());

    return sb.toString();
}
```

The above code is an internal method that creates the SQL text for an INSERT command. There are similar methods for SELECT, DELETE and UPDATE, but we have chosen this one since it is a good example of using reflection.

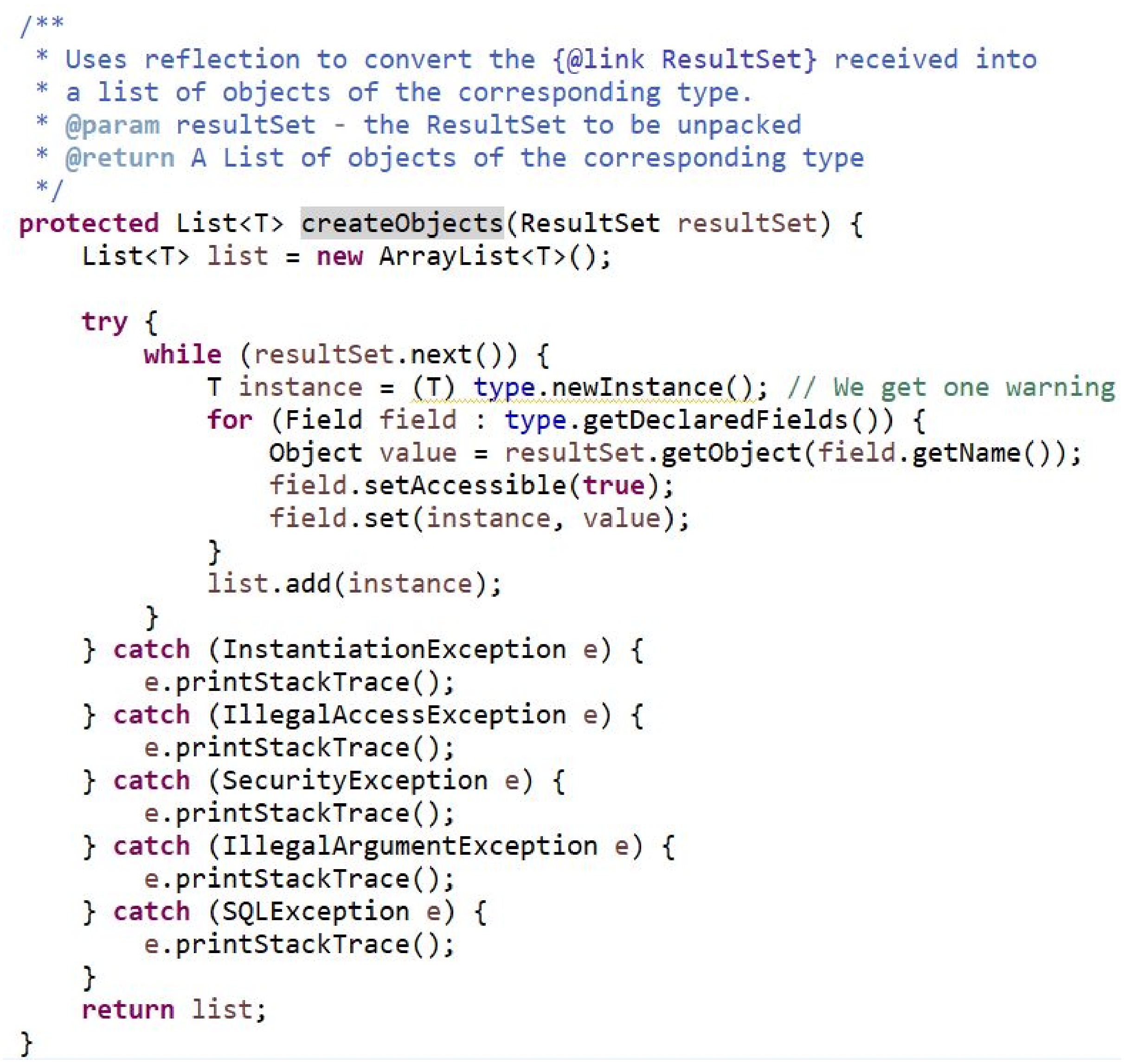
The program obtains the names of the attributes of the data we are working with, then relies on the convention that table columns have exactly the same name as the entity attributes to compose the SQL command. The final result is something like `INSERT (id, email) INTO User VALUES (?, ?)`, where the question marks will later be replaced in code with the actual values to be inserted.



```java
/**
 * Finds an element of the corresponding type in the database
 * given its id. If no elements are found with the given id,
 * this method will return null.
 * @param id - the id of the element to be searched for
 * @return The element, if found, otherwise null
 */
public T findById(int id) {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    String query = createSelectQuery("id");
    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.setInt(1, id);
        resultSet = statement.executeQuery();

        List<T> elems = createObjects(resultSet);
        return elems.isEmpty() ? null : elems.get(0);
    } catch (SQLException e) {
        LOGGER.log(Level.WARNING, type.getName() + "DAO:findById " + e.getMessage());
    } finally {
        ConnectionFactory.close(resultSet);
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }
    return null;
}
```

The above is the function that retrieves an item given its id. The steps are simple: open a connection, put the SQL text generated with a helper function in a PreparedStatement, set the required values into it (this is the step that replaces question marks with actual values) and execute it.

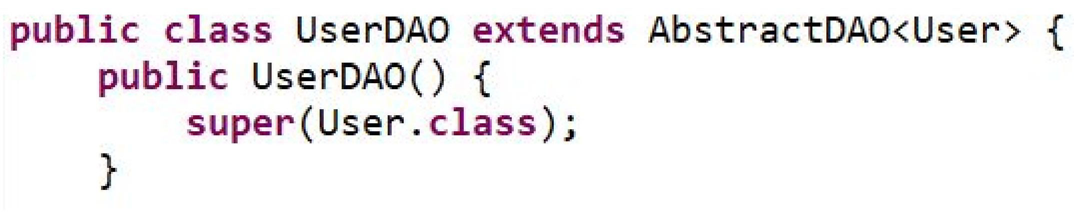
Interesting to note is that we call return inside the try-catch block. In this case, the finally block *still executes*, which means that even if we return a value, the used resources will still be closed.



```java
/**
 * Uses reflection to convert the {@link ResultSet} received into
 * a list of objects of the corresponding type.
 * @param resultSet - the ResultSet to be unpacked
 * @return A List of objects of the corresponding type
 */
protected List<T> createObjects(ResultSet resultSet) {
    List<T> list = new ArrayList<T>();

    try {
        while (resultSet.next()) {
            T instance = (T) type.newInstance(); // We get one warning
            for (Field field : type.getDeclaredFields()) {
                Object value = resultSet.getObject(field.getName());
                field.setAccessible(true);
                field.set(instance, value);
            }
            list.add(instance);
        }
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (SecurityException e) {
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return list;
}
```

One of the most important functions is the helper function above. We have to admit that it is rather poorly written, because we do not treat the many possible exceptions in any way.

This helper function converts the ResultSet obtained from executing a query into a usable list of items. It is declared protected so that it cannot be accessed from the outside, but subclasses of AbstractDAO can still make use of it.



```java
public class UserDAO extends AbstractDAO<User> {
    public UserDAO() {
        super(User.class);
    }
```

The above code is from the UserDAO class. Notice how simple it is to pass along the type of the class we will handle to the AbstractDAO.

```
/**
 * Finds a user based on the given e-mail address.
 * Note that this method assumes that there is at
 * most one user with this e-mail. Otherwise, the
 * first one encountered will be returned. If no
 * matching users are found, this method returns null.
 * @param email - the e-mail to search by
 * @return The user, if found, otherwise null
 */
public User findByEmail(String email) {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    String query = createSelectQuery("email");
    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.setString(1, email);
        resultSet = statement.executeQuery();

        List<User> users = createObjects(resultSet);
        return users.isEmpty() ? null : users.get(0);
    } catch (SQLException e) {
        LOGGER.log(Level.WARNING, "ItemDAO:findByName " + e.getMessage());
    } finally {
        ConnectionFactory.close(resultSet);
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }
    return null;
}
```

For comparison with the AbstractDAO method, the method above finds a user based on e-mail. One can see that the similarities with the generic method are very strong. Using generics allows us to avoid duplicating code like this, because it handles all the common cases. We need special code for the above only because we did not strive to make AbstractDAO even more generic, as we will see in chapter 6.

4.2. Managers

The manager classes do not use generics. As a result, there is a dose of code duplication among them, which is a good target for improvement in the future.

```java
public void delete(OrderItemDTO dto) {
    dao.delete(dtoToElem(dto));
}
```

Managers mostly just forward their responsibilities to an aggregated DAO object. As a result, we will not focus on the DAO-like functionalities that they provide.

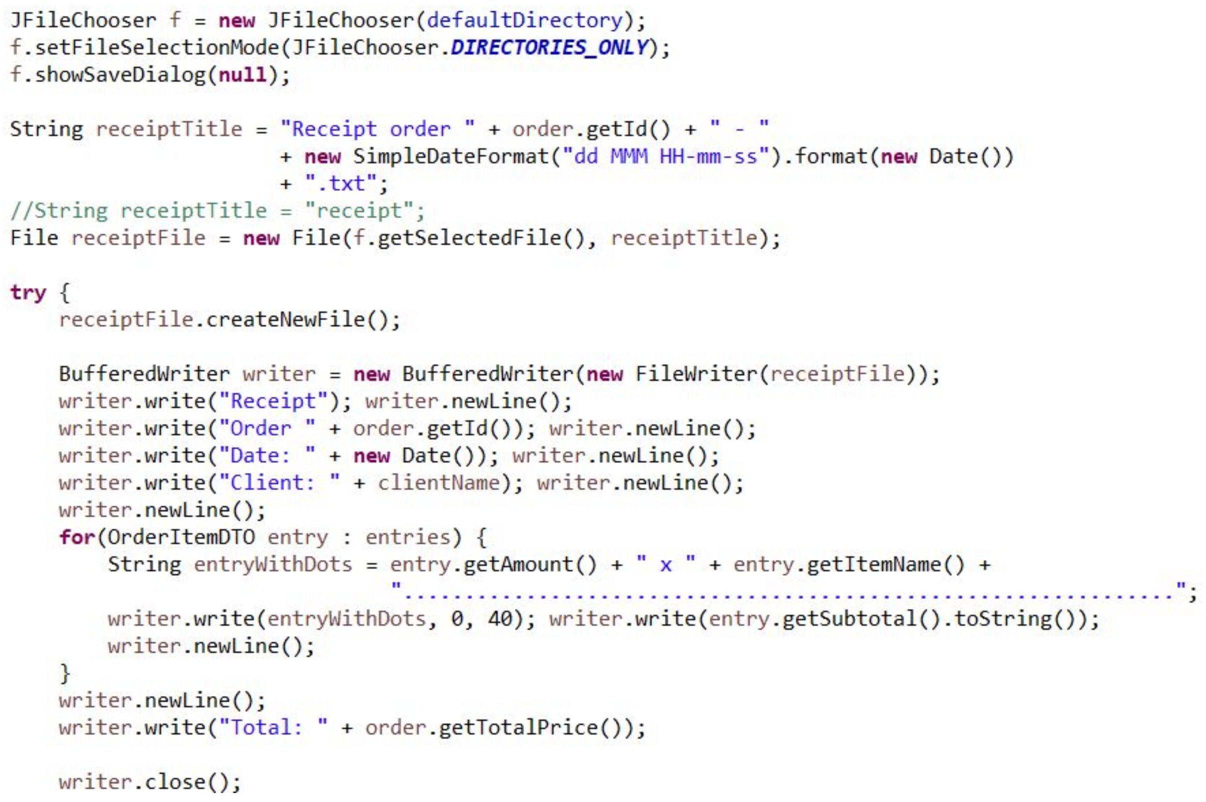
```java
/**
 * Converts an element to a DTO.
 * @param oi - the OrderItem to convert
 * @return An OrderItemDTO with the relevant data
 */
private OrderItemDTO elemToDto(OrderItem oi) {
    OrderItemDTO dto = new OrderItemDTO();
    // Transmitted
    dto.setId(oi.getId());
    dto.setItemId(oi.getItem_id());
    dto.setOrderId(oi.getOrder_id());
    dto.setAmount(oi.getAmount());

    // Computed
    Item i = iDao.findById(oi.getItem_id());
    dto.setItemName(i.getName());
    dto.setSubtotal(i.getPrice()
                    .multiply(new BigDecimal(oi.getAmount())));

    return dto;
}
```

The core functionality of the data managers is actually their ability to convert DTOs to entities and entities to DTOs. Since DTOs were designed to store strictly more information than the underlying entities, we will not show the dtoToElem method, but will instead present this one.

Notice how most attributes are simply copied over from the entity, but “item name” and “subtotal” do not exist in an OrderItem object, so they must be obtained here. It is for this reason that a data manager may sometimes need more than one DAO and it is for this reason that we choose not to simply display the entities, which would be far simpler.

4.3. ReceiptWriter

We have, indeed, succeeded at attaining the complete functionality of composing the receipt in a single class.



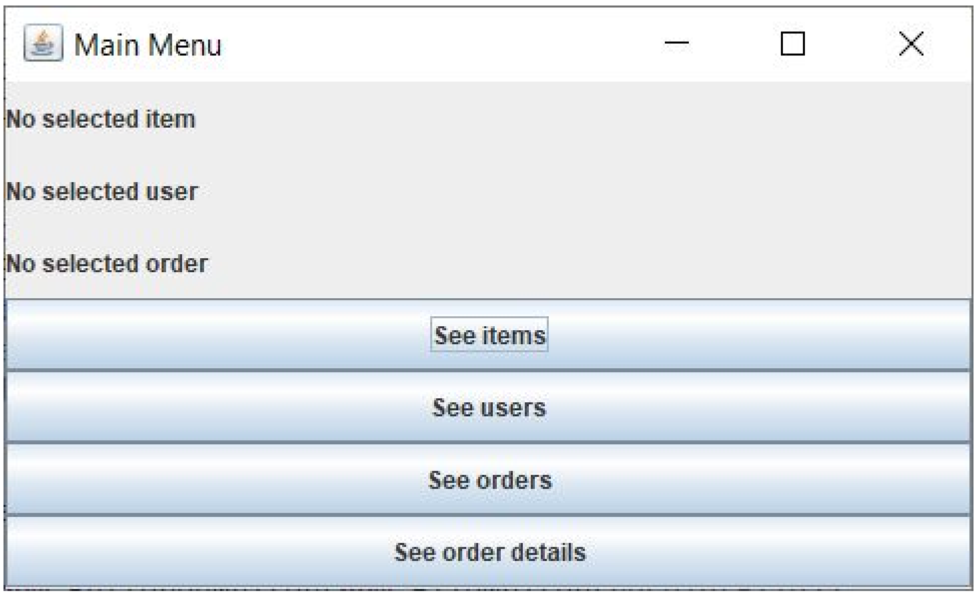
```java
JFileChooser f = new JFileChooser(defaultDirectory);
f.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
f.showSaveDialog(null);

String receiptTitle = "Receipt order " + order.getId() + " - "
                    + new SimpleDateFormat("dd MMM HH-mm-ss").format(new Date())
                    + ".txt";
//String receiptTitle = "receipt";
File receiptFile = new File(f.getSelectedFile(), receiptTitle);

try {
    receiptFile.createNewFile();

    BufferedWriter writer = new BufferedWriter(new FileWriter(receiptFile));
    writer.write("Receipt"); writer.newLine();
    writer.write("Order " + order.getId()); writer.newLine();
    writer.write("Date: " + new Date()); writer.newLine();
    writer.write("Client: " + clientName); writer.newLine();
    writer.newLine();
    for(OrderItemDTO entry : entries) {
        String entryWithDots = entry.getAmount() + " x " + entry.getItemName() +
                               "...............................................................";
        writer.write(entryWithDots, 0, 40); writer.write(entry.getSubtotal().toString());
        writer.newLine();
    }
    writer.newLine();
    writer.write("Total: " + order.getTotalPrice());

    writer.close();
```

The above is a fragment of code containing most of this functionality. There are a few simple steps:

- We create a JFileChooser that defaults to a directory inside our project designed specifically for the receipts
- We then create a dialog where the user chooses exactly where to save the receipt (starting from the default directory)
- We give the receipt a unique title via timestamping and create it
- We then use a buffered file writer to write into the new file, so that we get a nicely-formatted receipt. The result can easily be seen by running the project code.

4.4. Views

We indeed got to have three view classes and five total view objects in the application. All of the views are subclasses of JFrame (which, in hindsight, we should have made subclasses of JPanel for better flexibility). We will present them, with images, one by one, but we will not focus very much on the code itself - rather describing it in words - because it is bulky but not very informative.

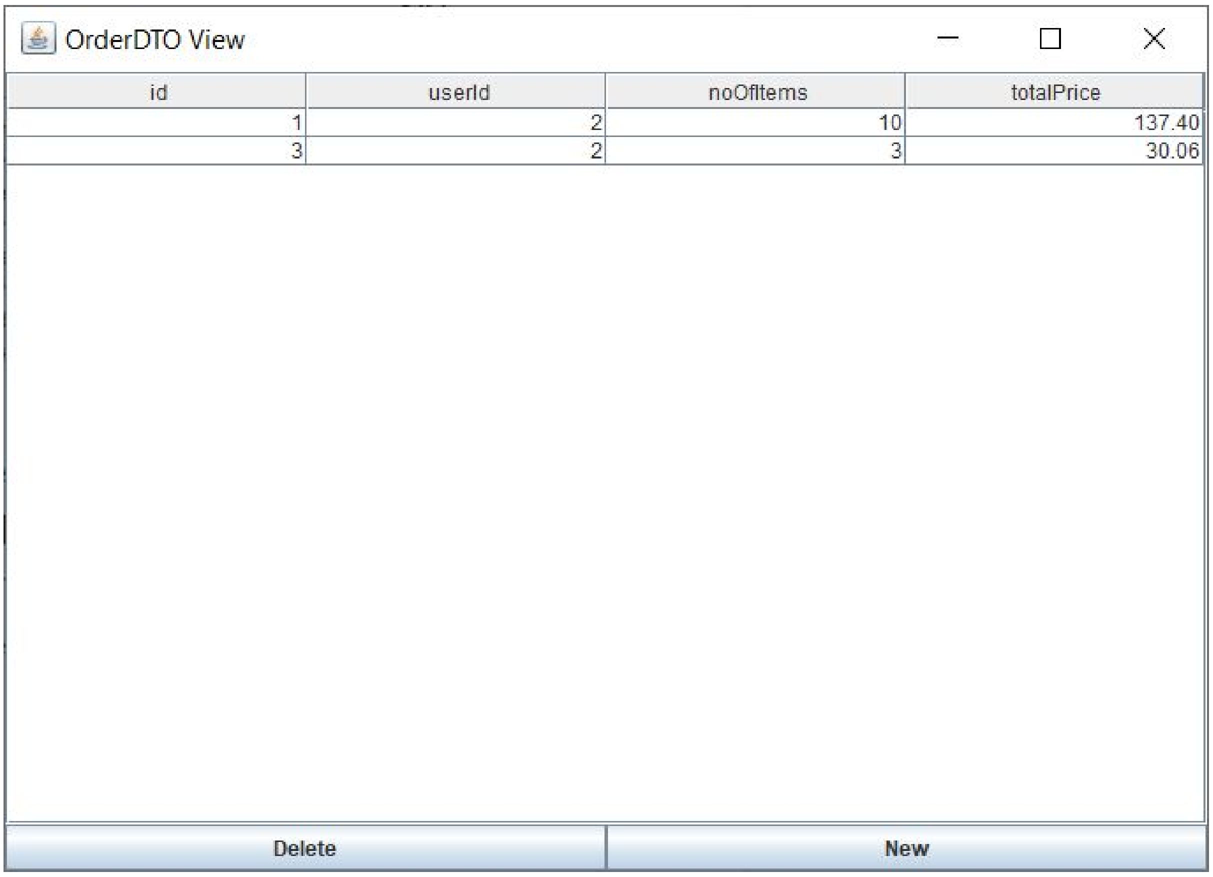


Pictured above is the main program window (a MainView). The program terminates only when this window is closed. It has three labels to display the currently selected item, user and order, and four buttons.

The first three buttons each open a window (a GenericView) showing respectively items, users and orders. To prevent needlessly repeating costly database access operations, the data in these views is only fetched once at the start of the program, then kept in sync with the database as the program runs.

Finally, the last item only works if there is a currently selected order. If there is, then the button opens a window (a CartView) where the more specific details of the order (such as which items were ordered and in what quantity) are displayed.

The MainView provides methods to set its labels from the controller, including an all-purpose method that takes one parameter saying *what* to display on the label, and another parameter saying *which* label to display on. We will see this employed in the Controller.


OrderDTO View

id	userId	noOfItems	totalPrice
1	2	10	137.40
3	2	3	30.06

The above is an example of GenericView. It is, as expected from its name, a view that can accomodate any kind of elements and display them in a table (in our case, OrderDTOs - notice how there is a column for total price!).

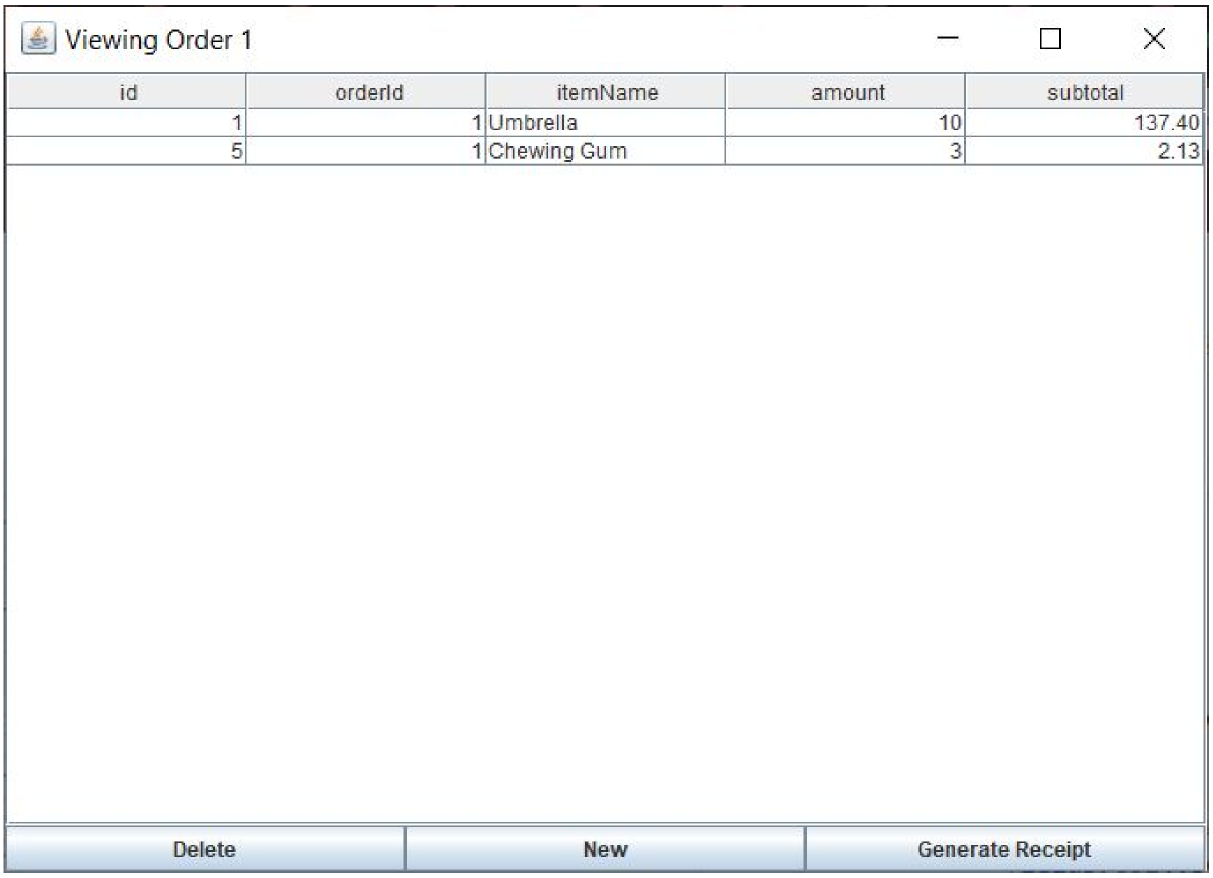
Its two buttons allow a user to delete and insert data, while for updating data the user can simply double-click a cell and change its value. Note that the id field cannot be edited and changing DTO-specific data such as the total price doesn’t make sense, and thus creates no effect.

Once again, this functionality will be explored further in the upcoming subchapters, since the view itself does not handle button events, but merely passes them on.

The most important part of the GenericView is that it includes a private class - the GenericTableModel, which is a subclass of Swing’s default table model. This is actually the class that handles the data, while the GenericView is more of a wrapper for the table.

```
public Object getValueAt(int row, int col) {
    Object result = new Object();

    boolean temp = fields[col].isAccessible();
    fields[col].setAccessible(true);
    try {
        result = fields[col].get(data.get(row));
    } catch (IllegalArgumentException e) { // TODO: handle catch blocks, as always!
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
    fields[col].setAccessible(temp);

    return result;
}
```

The above is a snippet of code from the table model. It stores its data in a List of type T, and maintains a list of Field objects describing the attributes of T (these coincide with the table columns). To access the value at a certain cell, we locate the Field corresponding to the column, then use it to extract the correct attribute from the row-th element of our data list.



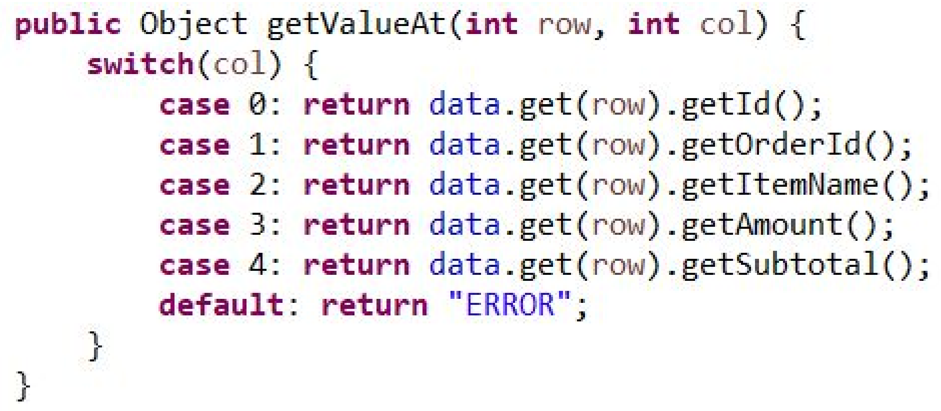
id	orderId	itemName	amount	subtotal
1	1	Umbrella	10	137.40
5	1	Chewing Gum	3	2.13

This is the CartView, showing detailed information about Order 1 (as we can see in the window title). Its presentation is quite similar to GenericView and it shares some functionality with it, such as being able to edit certain data (in this case, only the amount column is editable) inline and in its deletion function.

To open a such window, the user must select an order by clicking it in the order view, then click “See Order Details” in the main menu window. This will cause a CartView detailing the chosen order to appear. This CartView is generated only when needed, since preparing one for each Order in advance would take a prohibitive amount of time and memory (at least for situations with 10-20 orders of 10-20 items each).

There are two importantly different functionalities of the CartView relative to the GenericView. The first one is the “New” functionality. In order for the “New” button to work, the user must first select an item in the item view. Then, when clicking “New”, a dialog pops up, asking how many items should be added to the order. This is where the validation happens - if the user gives a value that is too high, they will get an “under stock” error and the items will not be added to the order.

The other different functionality is the “Generate Receipt” one. When a user wants to obtain a standalone representation of the order being viewed, they can click this button. A dialog will pop up, as per ReceiptWriter’s implementation, and the user will choose where to save the receipt.

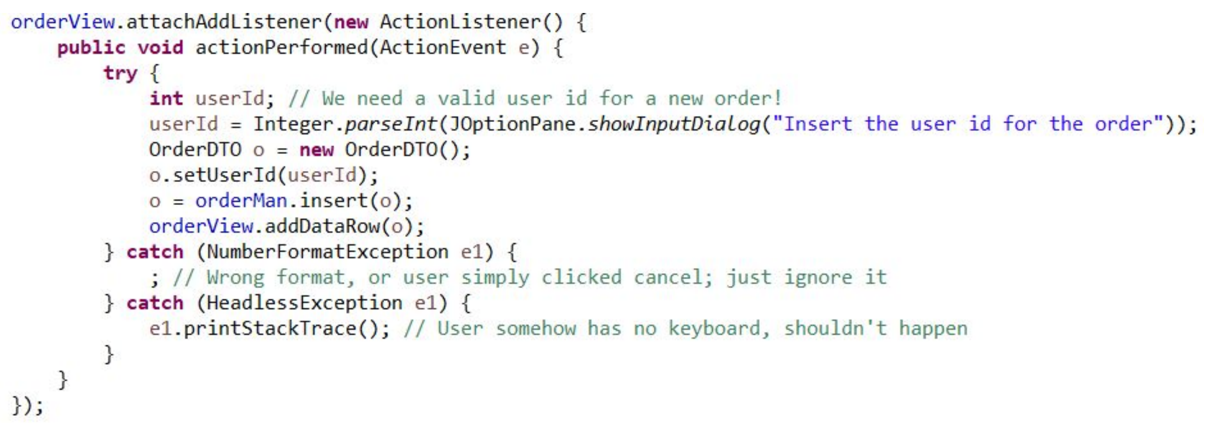


```java
public Object getValueAt(int row, int col) {
    switch(col) {
        case 0: return data.get(row).getId();
        case 1: return data.get(row).getOrderId();
        case 2: return data.get(row).getItemName();
        case 3: return data.get(row).getAmount();
        case 4: return data.get(row).getSubtotal();
        default: return "ERROR";
    }
}
```

This is the CartView implementation of getValueAt, without using generics. It is visibly simpler and easier to understand, but is far harder to scale across many fields and many classes, which is why we opted for generics whenever they were readily accessible.

4.5. Controller

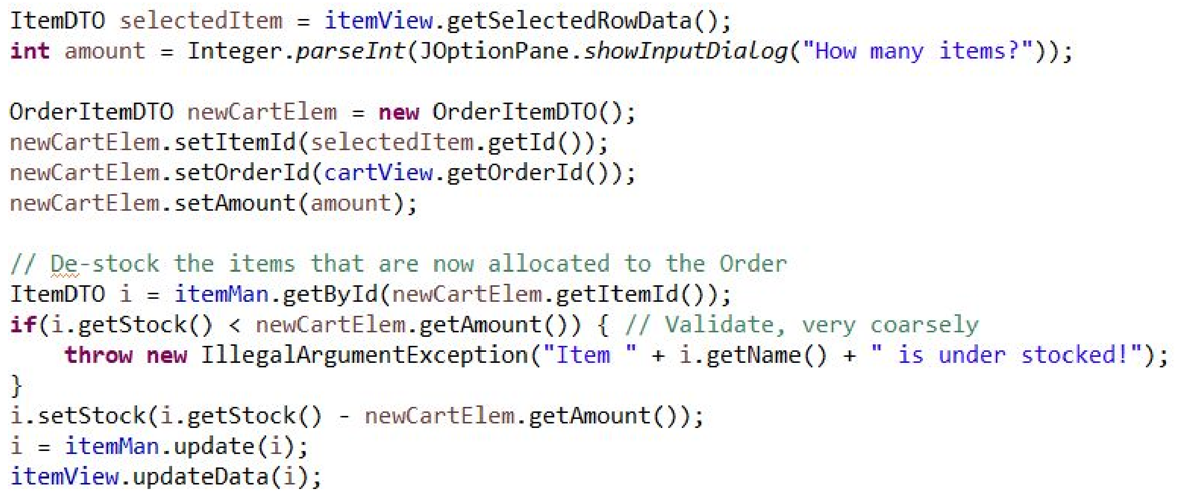
We ultimately created a pseudo-monolithic controller, in the sense that while there is a unique controller class, we have also created three listener classes that the controller instantiates and assigns to the views, which take upon themselves part of the controller’s tasks.



```java
orderView.attachAddListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            int userId; // We need a valid user id for a new order!
            userId = Integer.parseInt(JOptionPane.showInputDialog("Insert the user id for the order"));
            OrderDTO o = new OrderDTO();
            o.setUserId(userId);
            o = orderMan.insert(o);
            orderView.addDataRow(o);
        } catch (NumberFormatException e1) {
            ; // Wrong format, or user simply clicked cancel; just ignore it
        } catch (HeadlessException e1) {
            e1.printStackTrace(); // User somehow has no keyboard, shouldn't happen
        }
    }
});
```

The main role of the controller is to properly set up all of the various listeners. While some of these tasks are delegated to the listener classes, more specific activities are treated with anonymous listeners, such as the above.

The above code describes what happens when a user wants to add a new Order: they are requested to provide a user id, since orders are bound to their users in the database through a foreign key constraint (so it would be impossible to have an order with an invalid user, because even on user deletion, we cascade the change). Once the user id is provided, we update the database and the view with the new data.

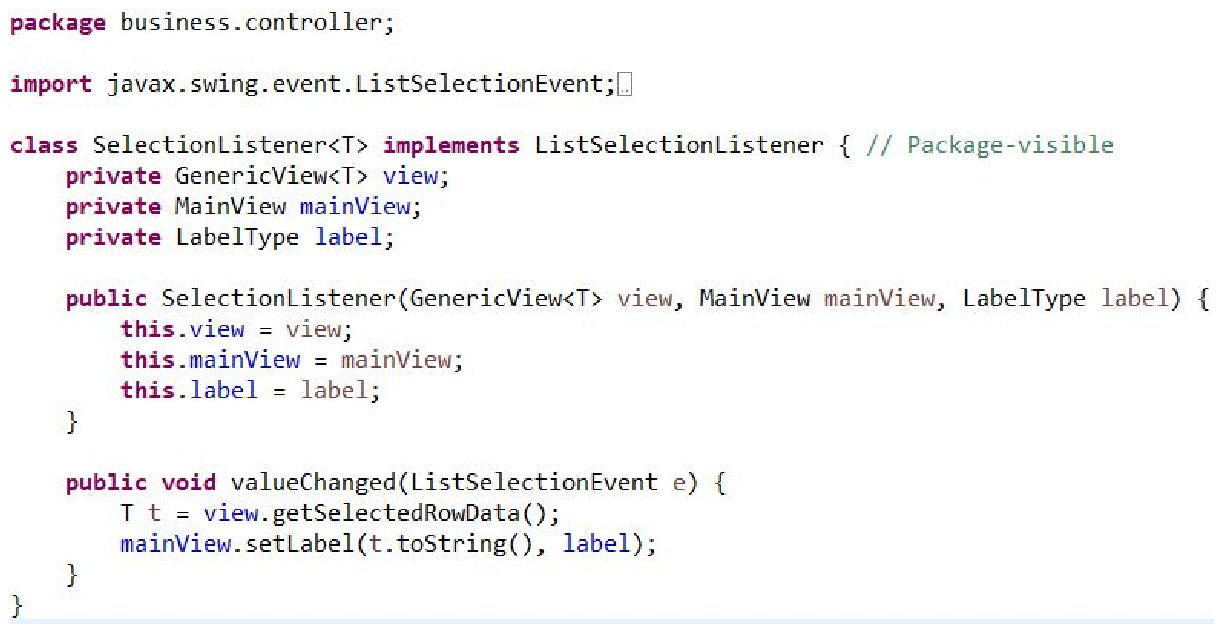


```java
ItemDTO selectedItem = itemView.getSelectedRowData();
int amount = Integer.parseInt(JOptionPane.showInputDialog("How many items?"));

OrderItemDTO newCartElem = new OrderItemDTO();
newCartElem.setItemId(selectedItem.getId());
newCartElem.setOrderId(cartView.getOrderId());
newCartElem.setAmount(amount);

// De-stock the items that are now allocated to the Order
ItemDTO i = itemMan.getById(newCartElem.getItemId());
if(i.getStock() < newCartElem.getAmount()) { // Validate, very coarsely
    throw new IllegalArgumentException("Item " + i.getName() + " is under stocked!");
}
i.setStock(i.getStock() - newCartElem.getAmount());
i = itemMan.update(i);
itemView.updateData(i);
```

The above is a section of the code for the CartView's "New" button listener. It opens a dialog to get the amount, then removes the items from stock if all is good. If not, we throw an exception which is caught later in the code, causing the error dialog to appear.

4.6. Listeners

We implemented a total of three listener classes: ChangeListener, which tracks when changes happen in the table and updates the database, DeleteListener, which activates when one of the GenericViews deletes an element, removing it from the view and from the database, and SelectionListener, which monitors which items get selected in the views and updates the labels in the MainView. They are all rather small and straight-forward, so we will only show one.



```java
package business.controller;

import javax.swing.event.ListSelectionEvent;

class SelectionListener<T> implements ListSelectionListener { // Package-visible
    private GenericView<T> view;
    private MainView mainView;
    private LabelType label;

    public SelectionListener(GenericView<T> view, MainView mainView, LabelType label) {
        this.view = view;
        this.mainView = mainView;
        this.label = label;
    }

    public void valueChanged(ListSelectionEvent e) {
        T t = view.getSelectedRowData();
        mainView.setLabel(t.toString(), label);
    }
}
```

Note how this is also a generic class - the listener does not know, when compiled, exactly which label in MainView it will need to set, so we pass it a LabelType (which is an enumeration of ITEM, USER or ORDER) and make use of the catch-all *setLabel()* function in MainView to allow us not to have to select a specific label setting function at compile time.

5. Results

Based on extensive test runs of the application, its performance has proven acceptable, its interface manageable enough and no errors have been noticed. The program has not been unit-tested in any way, but since the main issues that could arise in a such program are related to the UI, unit-testing would not be the best way to go about it anyway

6. Conclusions

I believe that I have learned quite a bit about reflection by working on this project. It was a different experience to write code that is so general and applies to almost anything.

Overall, I am fairly pleased about the result of my work. However, there are still many avenues for possible improvement, of which I will list a few:

- I have not managed to perform the dependency inversion between the controller and the view; although I know how it should be done (define an interface in the controller package and have the view implement it), the result would be a very big interface, so I did not attempt it
- Likewise, it would be good if the DAOs passed through an interface, so that they could be changed to a totally different implementation (even by just swapping out our DAO module for a different one!) without any change in the code at all.
- A different idea for the display would be to put the item, user, order and orderItem views in a 2-by-2 grid display, so that the user can see all of them at once without having to keep swapping windows. This does have the slight inconvenient of forcefully removing the MainView, but I believe it could make for a better UI once implemented.
- It would be wise to refactor the views so that they are subclasses of JPanel (with the enclosing JFrames created elsewhere, for example in the constructor) rather than of JFrame. This would

carry the benefit of allowing them to be embedded into other windows - for example, perhaps it would be possible to display OrderItems with the same generic display the other classes use, separate from the fact that the buttons and functionality of the window differ otherwise.

- If the code grows, it may be wiser to split the controller into multiple small classes, each watching its own view. Using something like the Observer pattern, it would be possible to have these classes still notify each other of relevant changes, without forcing them to become a single class.
- It is probably possible to remove most of the code duplication in the data managers by employing inheritance and generics, much like for DAOs.

7. Bibliography

While no particular source outside of the school materials has been a great inspiration, the following sites were useful now and then for overcoming impediments:

- https://db.apache.org/ojb/docu/guides/jdbc-types.html (contains a map of Java types to MySql types, which was very useful in deciding what types my entity attributes should have)
- https://stackoverflow.com (although I have never posted a question on StackOverflow, seeing questions others have asked often gave me an idea of how to approach my own challenges)
- http://tutorials.jenkov.com/java-internationalization/simpledateformat.html (I needed a way to uniquely identify every receipt, and using Date objects seemed like a simple solution at first)