# Urban Voting System

Timofei Kuzin

May 2024

## 1 Main idea

### 1.1 Introduction

The ”Urban Environment Voting” project introduces a platform enabling citizens to influence the development and enhancement of their urban surroundings directly. This initiative allows for collective decision-making on what people want to see in their city.

### 1.2 Objective

Options could range from transforming an abandoned space into a library, community center, or museum to building a new volunteer center. The best thing about this is that residents can change their cities and participate in making the best life for them.

### 1.3 Process

The process is streamlined and accessible, involving the following steps:

1. Registration on the platform to ensure a secure and verified voting process.
2. Navigation to the ”Menu” section to view active projects.
3. Review of proposed ideas, with detailed descriptions on cards.
4. Voting on the preferred project by choosing the card.
5. Compilation and announcement of the voting results, followed by the initiation of the project receiving the highest community endorsement.

### 1.4 Impact

This application empowers residents to play an active role in urban development, promoting a more democratic and community-focused approach to city planning. The project also underscores the importance of citizen participation in decision-making.

## 2 App Logic

### 2.1 Overview

Logic of this app is separated by packages so the code is more readable. Here is a diagram for all classes but then I will put diagrams by every package to make the documentation easy to read.

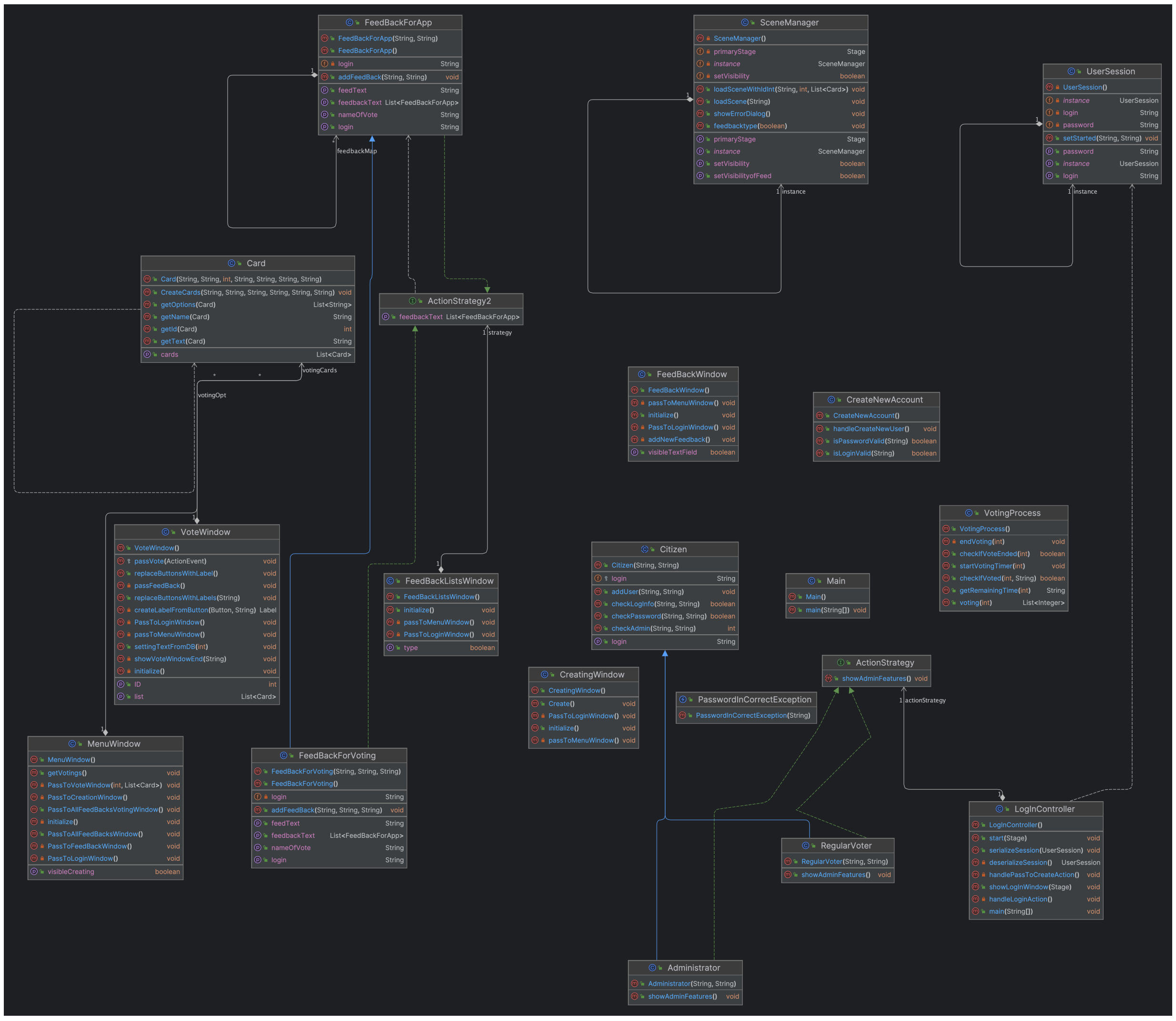


Figure 1: A diagram for all classes.

### 2.2 Citizen Logic

In this package, we have classes 'Citizen', 'UserSession', 'RegularVoter', 'Administrator', and an interface 'ActionStrategy'. So by this diagram, we can see that Administrator and RegularVoter are implementing the ActionStrategy interface. The reason for that is to switch the logic of the app for admin and regular users, cause the admin will have more features such as creating new voting and checking all feedback on the app and voting. The switching of this strategy pattern takes place in the LogInController GUI class. So the Citizen has some methods to check if the user is admin by just checking a column in the database. Also, there is a method to check login info, check if the password is correct, and if it is not then my exception will be thrown, and also a method to add a new user to the system.

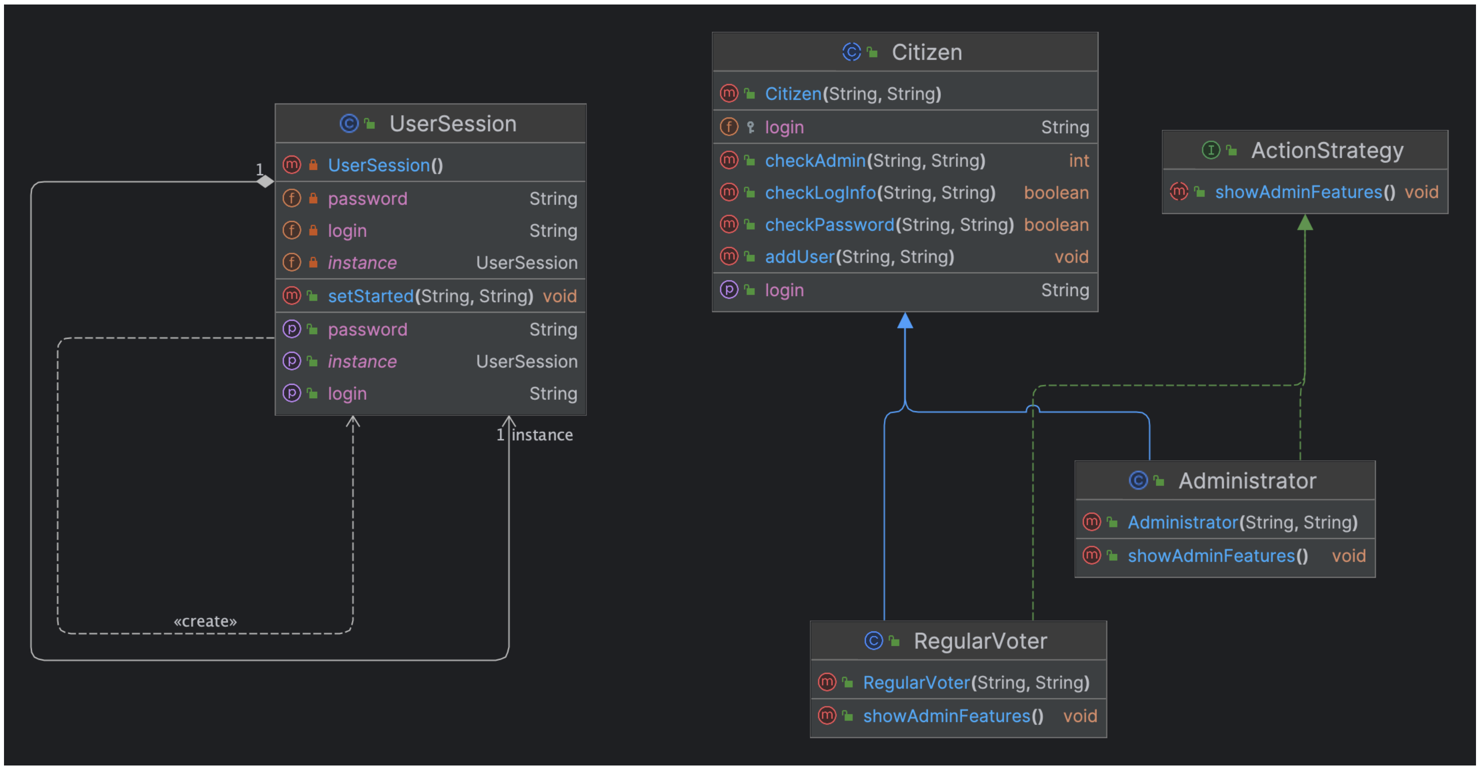


Figure 2: A diagram for Citizen package.

### 2.3 Database Logic

In this package we have classes 'DataBaseConnection', 'DataBaseConnection.DataBaseInterface', and an exception 'PasswordInCorrectException'. DataBaseConnection class is used only for the purpose to connect to database so we can send requests and the main logic of all requests is in the class DataBaseConnection.DataBaseInterface. These classes are only used to separate SQL requests from application logic, so the code will be more readable. Also in this package, there is an exception class for incorrect passwords. This exception is thrown when the user sends a login that is in the database but the password is not correct for this login. Lots of classes from the backend of the app are using the DataBaseConnection.DataBaseInterface class to communicate with the database and store main information about all of the users interacting with an app.

## 2.4 Feedbacks logic

In this package, we have classes 'FeedBackForApp', 'FeedBackForVoting' and an interface 'ActionStrategy2'. So these classes are used to handle adding feedback to the database and getting them from the database. FeedBackForVoting is extended from FeedBackForApp. The differences between these two classes are that they use two different database tables and also the FeedBackForVoting has a String parameter for the name of voting.

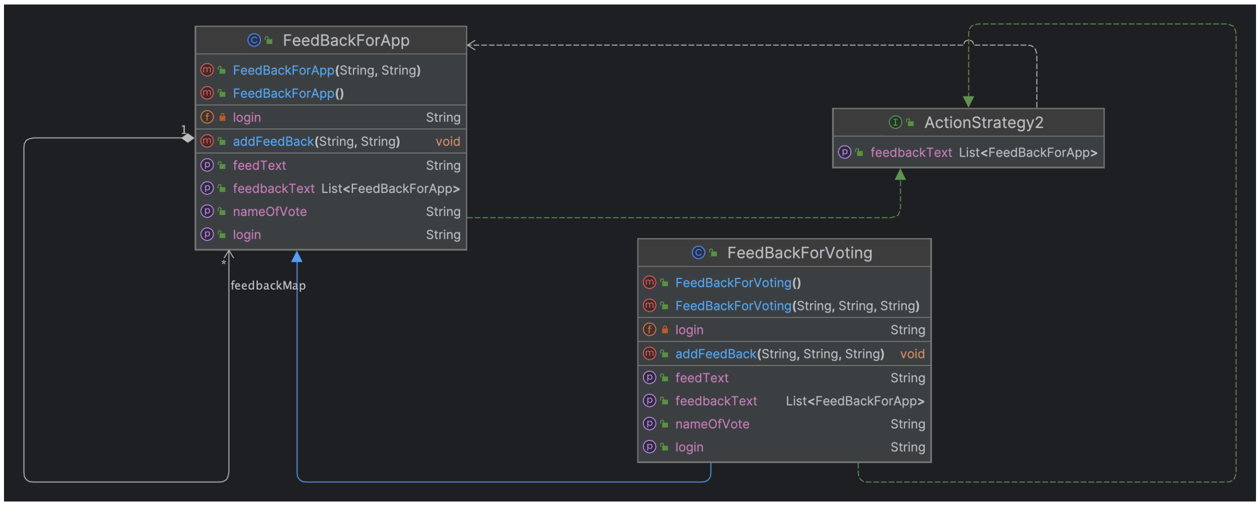

FeedBackForApp
FeedBackForApp(String, String)
FeedBackForApp()
login String
addFeedBack(String, String) void
feedText String
feedbackText List<FeedBackForApp>
nameOfVote String
login String
1
*
feedbackMap
ActionStrategy2
feedbackText List<FeedBackForApp>
FeedBackForVoting
FeedBackForVoting()
FeedBackForVoting(String, String, String)
login String
addFeedBack(String, String, String) void
feedText String
feedbackText List<FeedBackForApp>
nameOfVote String
login String

Figure 3: A diagram for Feedbacks package.

## 2.5 Vote Cards Logic

In this package, we have only one class 'Card'. It represents the voting card with all information including the name of voting all four options and information about this voting. Also in this method, there is a method to get all cards from the database and a method to create a new one.

## 2.6 Voting Logic

In this package, we have only one class 'VotingProcess'. That is a complicated class that works with a database and with a multithreading to handle the creation of lots of timers for every vote. So this class handles the vote option from the user, adding to the database information that this user voted for this voting and also adding to a database that this vote ended when the timer goes off.

## 2.7 GUI Logic

In this package, we have lots of classes that are loading fxml screens and handling all the actions. So the GUI in my project is separated from the main logic as supposed to be. The main classes in the GUI logic are the 'LogInController', 'SceneManager', and the 'MenuWindow'. From menu, every user can select for what voting will he vote and send feedback on the entire app, and the admin can create new voting and check all feedback. The LogInController is a start window for entire app. It switches the strategy pattern for users. And the SceneManager is kinda like a simple router that loads all the screens by their names.

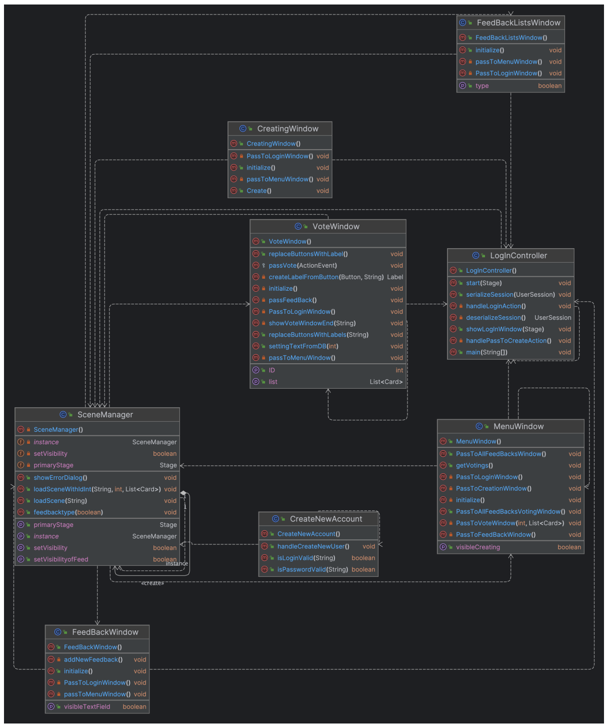
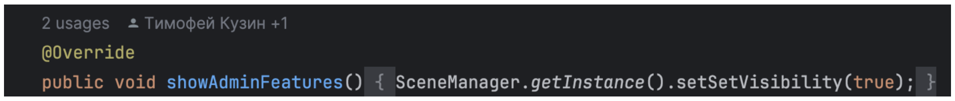


Figure 4: A diagram for GUI package.

## 3 Main criteria

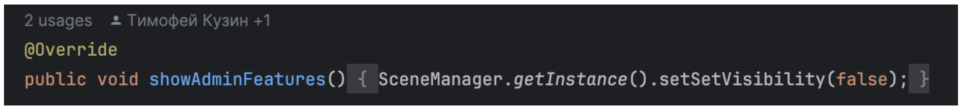
### 3.1 Inheritance

So as you can see by the diagrams there are two hierarchies of inheritance in the project. The first one is the 'Citizen', 'RegularVoter', and an 'Administrator' (Citizen Logic). The second one is 'FeedBackForApp' and a 'FeedBackForVoting' (Feedback Logic) So the polymorphism is also used in both hierarchies. Here is the admin method to set the visibility of admin features as true.



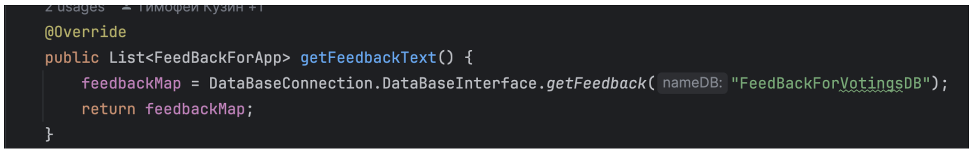
```
2 usages  Тимофей Кузин +1
@Override
public void showAdminFeatures() { SceneManager.getInstance().setSetVisibility(true); }
```

And here is the method in the 'RegularVoter' class that is used for the same purposes but sets visibility as false.



```
2 usages  Тимофей Кузин +1
@Override
public void showAdminFeatures() { SceneManager.getInstance().setSetVisibility(false); }
```

So the 'FeedBackForVoting' also has polymorphism for the function that sends a request for database.



```
2 usages  Тимофей Кузин +1
@Override
public List<FeedBackForApp> getFeedbackText() {
    feedbackMap = DataBaseConnection.DataBaseInterface.getFeedback( nameDB: "FeedBackForVotingsDB");
    return feedbackMap;
}
```

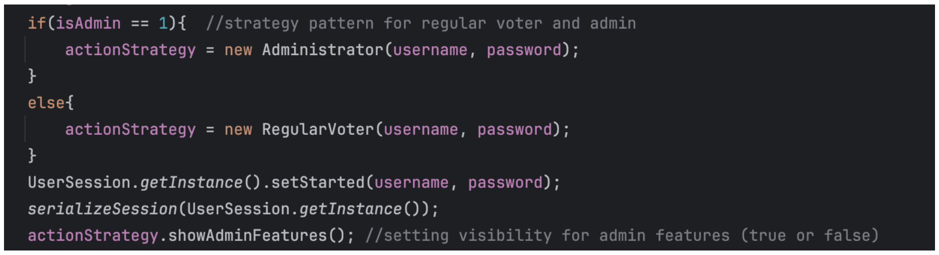
### 3.2 Other main criteria

So the aggregation is used in lots of places in code the perfect example is in the MenuWindow code where it sets a list of voting cards and then uses it to display all the voting that are available. Incapsulation is used in all places where it could be used to provide safety of code. So all the GUI logic is separated from app logic by using GUI package.

## 4 Other criteria

### 4.1 Pattern 'Strategy' in Citizen logic

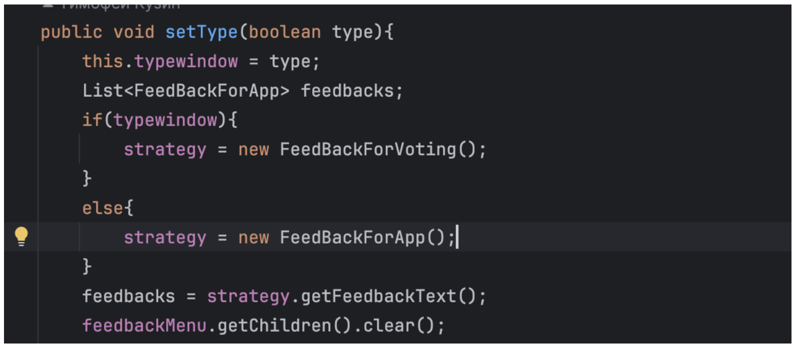
Classes 'Administrator' and 'RegularVoter' are implementing the ActionStrategy interface that is used to switch between two user classes. It starts from the 'LoginController' where the actionstrategy object is created and then the code checks if the user is an admin or not. Then the code creates a class 'RegularVoter' or 'Administrator' depending on the isAdmin variable. Then it uses a method of interface showAdminFeatures. This method will show admin features if the admin was logged in, if not then it will put false in the set visibility method of admin features. You can also see the functions that are in the classes here (Method in admin class) (Method in regular voter class).



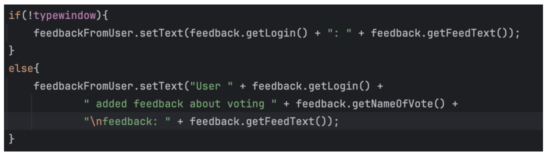
```
if(isAdmin == 1){  //strategy pattern for regular voter and admin
    actionStrategy = new Administrator(username, password);
}
else{
    actionStrategy = new RegularVoter(username, password);
}
UserSession.getInstance().setStarted(username, password);
serializeSession(UserSession.getInstance());
actionStrategy.showAdminFeatures(); //setting visibility for admin features (true or false)
```

## 4.2 Pattern 'Strategy' in FeedbackList logic

Classes 'FeedBackForApp' and 'FeedBackForVoting' are also implementing the other interface 'ActionStrategy2'. So it is created to show the list of feedback on one screen but if we want to see the feedback on the vote we have to put other information on the screen such as the name of the vote. So, the strategy is used in the FeedBackListsWindow. If the variable typewindow is false then all feedbacks on the app will be displayed, if true then there will be feedbacks on voting.



```
public void setType(boolean type){
    this.typewindow = type;
    List<FeedBackForApp> feedbacks;
    if(typewindow){
        strategy = new FeedBackForVoting();
    }
    else{
        strategy = new FeedBackForApp();
    }
    feedbacks = strategy.getFeedbackText();
    feedbackMenu.getChildren().clear();
```



```
if(!typewindow){
    feedbackFromUser.setText(feedback.getLogin() + ": " + feedback.getFeedText());
}
else{
    feedbackFromUser.setText("User " + feedback.getLogin() +
            " added feedback about voting " + feedback.getNameOfVote() +
            "\nfeedback: " + feedback.getFeedText());
}
```

## 4.3 Multithread and lambda function in VotingProcess class

This class is implemented multithreading to handle creating timers for all voting. So the main thing is the user clicks on some option in voting than goes some communication with the database and then the VotingProcess starts a random timer from 0 to 120 seconds, so the app will be tested more quickly. All timers are working with their own thread to handle lots of voting. Also in this class, there is a lambda function so if there will be a new object from 'Thread' it will already have a behaviour of timer. The lambda function configuring the Thread so it will have the role of timer.

## 4.4 Serialization in UserSession class

So the UserSession class implements Serializable interface. The main idea was to serialize the user login information if the user didn't logged out so that when the app starts previous user already will be logged in. Class user session is a singleton class which has all the login info. Serializing and deserializing methods are written in the LogInController GUI class that contains the main logic of serializing.

## 4.5 Exception

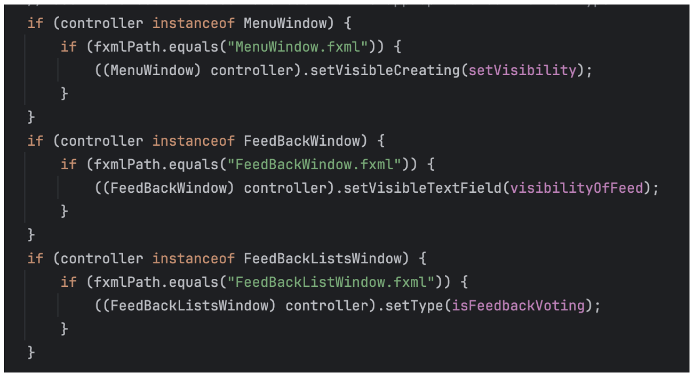
There is also a class PasswordInCorrectException. That is an exception that starts from DataBaseInterface class where it in the method 'checkPassword' checks the password and then if the password is not correct for this login the exception is thrown. The exception will be catched in LogInController class to display an error message that the password is incorrect.

## 4.6 RTTI in SceneManager

So there is also a construction 'instanceof' in the SceneManager but it's not really necessary there. It's just for double safety so there will be no problem while loading a new screen.

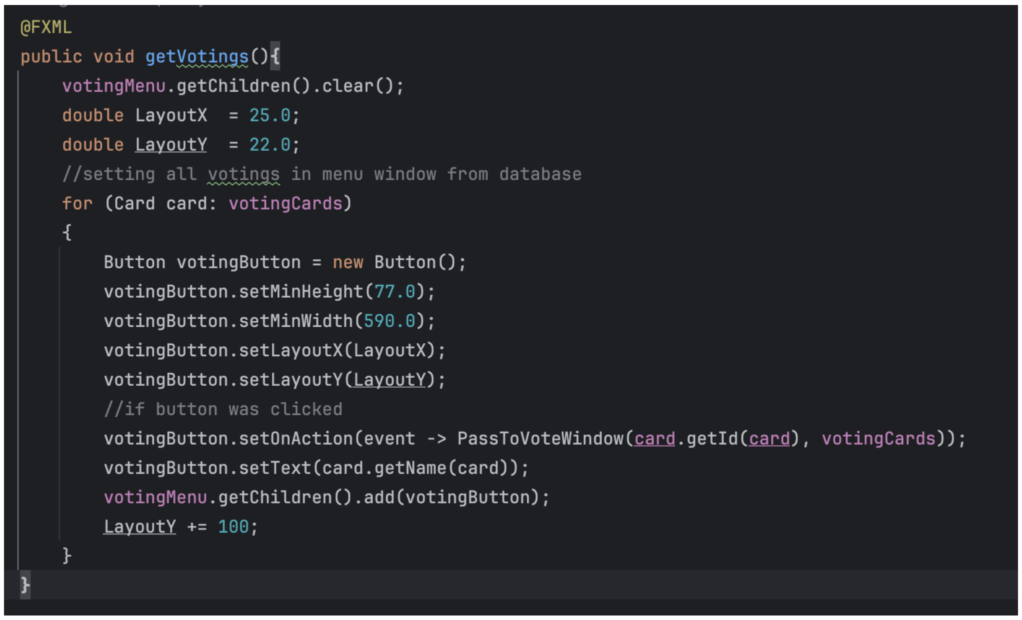
## 4.7 GUI

So the GUI is separated by the package from other parts of the app. It has automatically generated handlers by parameters "onAction" in fxml files, but also for example in MenuWindow class there is a dynamic setting of the button depends on how many available votings are in the database.



```java
if (controller instanceof MenuWindow) {
    if (fxmlPath.equals("MenuWindow.fxml")) {
        ((MenuWindow) controller).setVisibleCreating(setVisibility);
    }
}
if (controller instanceof FeedBackWindow) {
    if (fxmlPath.equals("FeedBackWindow.fxml")) {
        ((FeedBackWindow) controller).setVisibleTextField(visibilityOfFeed);
    }
}
if (controller instanceof FeedBackListsWindow) {
    if (fxmlPath.equals("FeedBackListWindow.fxml")) {
        ((FeedBackListsWindow) controller).setType(isFeedbackVoting);
    }
}
```

Figure 5: RTTI in scenemanager



```java
@FXML
public void getVotings(){
    votingMenu.getChildren().clear();
    double LayoutX  = 25.0;
    double LayoutY  = 22.0;
    //setting all votings in menu window from database
    for (Card card: votingCards)
    {
        Button votingButton = new Button();
        votingButton.setMinHeight(77.0);
        votingButton.setMinWidth(590.0);
        votingButton.setLayoutX(LayoutX);
        votingButton.setLayoutY(LayoutY);
        //if button was clicked
        votingButton.setOnAction(event -> PassToVoteWindow(card.getId(card), votingCards));
        votingButton.setText(card.getName(card));
        votingMenu.getChildren().add(votingButton);
        LayoutY += 100;
    }
}
```

Figure 6: Dynamic buttons in MenuWindow