

**Project report**

**Multiplayer racing game**

**Date**:

28.5.2017

**Made by:**

Alexandru Ungureanu (224469)

Dominik Lukac (239837)

Vladimir Rocin (239854)

**Supervisors:**

Ib Havn

Lars Sorensen

Richard Brooks

Table of figures

[Abstract 5](#_Toc452635970)

[1. Introduction 6](#_Toc452635971)

[2. Analysis 7](#_Toc452635972)

[2.1 Requirements 7](#_Toc452635973)

[2.2 Key features 9](#_Toc452635974)

[2.3 Constraints 10](#_Toc452635975)

[2.4 Risks 10](#_Toc452635976)

[2.5 Use case diagram 11](#_Toc452635977)

[2.6 Actor description 12](#_Toc452635978)

[2.7 Use case description 13](#_Toc452635980)

[2.8 Activity diagram 14](#_Toc452635981)

[2.9 Class diagram 15](#_Toc452635982)

[3. Design 16](#_Toc452635983)

[3.1 Design Class Diagram 16](#_Toc452635984)

[3.2 GUI design 18](#_Toc452635985)

[3.3 Sequence diagram 20](#_Toc452635987)

[4. Implementation 21](#_Toc452635988)

[4.1 Database 21](#_Toc452635989)

[4.2 Controller 26](#_Toc452635990)

[5. Testing 29](#_Toc452635991)

[6. Discussion 30](#_Toc452635992)

[6.1 AdminWindow 30](#_Toc452635993)

[6.2 RMI 30](#_Toc452635994)

[6.3 Stop/start election 30](#_Toc452635995)

[6.4 No history of votes 30](#_Toc452635996)

[7. Results 31](#_Toc452635997)

[8. Conclusion 32](#_Toc452635998)

[9. References 33](#_Toc452635999)

Abstract

*The goal of this project is to develop a real-time application based on freeRTOS which will run on AVR micro-controller. This include also design and implementation of protocol which describes communication between the AVR and PC via usb serial connection. The application has a form of the game for two players, in our case racing/avoiding obstacles.*

1. Introduction

When talking about real-time operating systems, the main notion that arises is the “deadline”. An application is considered to be real-time only in the case that all its tasks can meet the specified deadlines, i.e. that the system produces a response within a predefined amount of time. This topic is of big importance in the realm of embedded programming.

The purpose of this paper is to document the development of a real-time system, starting from the analysis phase, including detailed requirements specifications as well as use case modeling, continuing with the design phase, which describes the enities of the systems (i.e. tasks), planning of the shared resources and of the relation between tasks, and ending with the implementation and testing phase, which can be summarized by the application of the design and constant check of the requirements. It is important to note that the testing phase will in fact represent the proof that the developed system is in fact real-time, and thus all the tasks meet their deadlines without fail.

The essence of the application is a game developed for a game console which is powered by an ATMega324P. Supporting a serial connection, the application is thus a multiplayer game. It is decided that the system will be a racing game. The point of the game is having two cars controlled by two players which must avoid all of the obstacles in their way. If a car hits anything in front of it, it is destroyed and the player controlling the other car wins. Obstacles will be spawned at one end of the display and move towards the other, opposing the movement of the cars. A car would be represented by one pixel/diode and would be allowed to move freely around, while obstacles would move towards the cars and then out of screen once they get out of the boundaries of the display. The goal of the game is to last longer than the opponent. Additionally, the game will end once there is a winner in a 5 rounds game.

In the following sections, as mentioned before, the main parts of the development process will be documented and represented in detail. This will be followed by a discussion section where the most relevant difficulties will be presented and analysed. At the end, a conclusion will be present in order to sum up again the contents of this paper and its goal.

1. Analysis

The analysis phase is used for deciding the game rules, constraints and features. Here, the flow of the game is thoroughly planned and using the above mentioned concepts, the first documentation for the developed system is being made.

This phase includes the creation of the following:

* + A list of requirements
  + Use case diagrams
  + Use case descriptions
  + Activity diagram
  1. Requirements
     1. Functional requirements

The following list of requirements which shows everything the system will have to be capable of is based on the first meetings of the group but had also been modified through the development process as we found out, that some things can be done better or easier. The list is divided by the type of user and ordered by the importance of the requirements.

**Player 1 (on game console):**

* See the movement of the car on the display
* Move with the car to sides using joystick
* At the end of the game, display who won

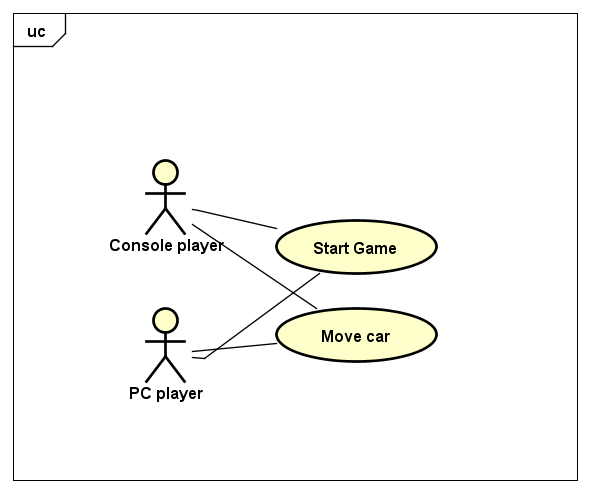
**Player 2 (on PC/Laptop):**

* Start/Load the game
* Play by using simple inputs (arrows) instead of the controller
* Can see the score, game time, progress of the game
* End/terminate the game
* At the end of the game, see who won

**Other**

* Cars can collide
* If a car makes a move towards an edge of the display it appears on the other side, but this move is possible only sideways.
* Movement into obstacle is not allowed but only counts as loss if it is frontal collision.
* Game results are displayed constantly on the PC console
  + 1. Non-funcitional requirements
* The system must be implemented with FreeRTOS
* Need to have our protocol with flow control and error detection
* Serial connection between game-console and PC
* Code should be unit tested
* We must use semaphores or mutexes
* We use only given hardware (ATmega 3247, DOT matrix, usb serial communication, joystick, R2R DA)
* PC app in C
* Use oscilloscope for computation time measurements
  1. Constraints
* Game is only for two players
* Application does not keep track of games before restart of the controller.
* No GUI
* Difficulty is fixed
  1. Use case diagram

Based on requirements the use case diagram shows all the functionalities that must be implemented. As it can be seen from the picture below, the use case diagram is very simplistic mainly because the developed system is a very simple game. Furthermore, the limitation of the control and display of the hardware render this diagram in the way it is presented.



*Figure 1 – use case diagram*

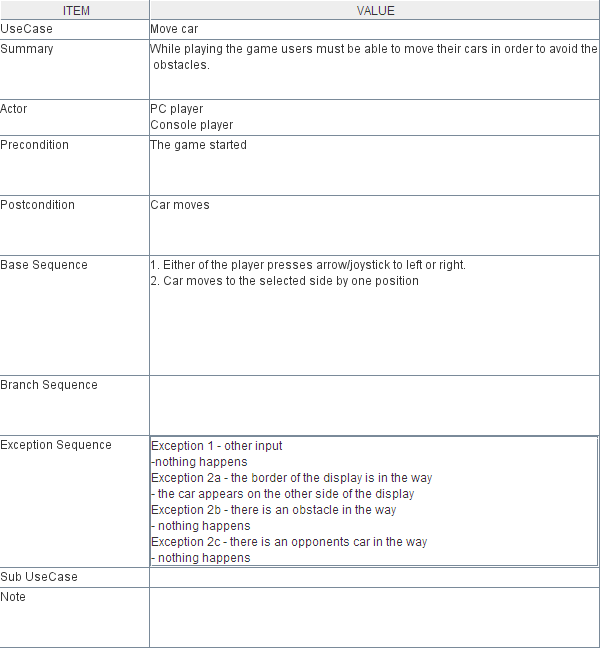
As it can be seen, there are two actors in the system, namely the two players. They both have the possibility of triggering the start of the game as well as to move the car the is associated with each of the players. Everything else that the system provides is not dependent on the users of the system.

* 1. Actor description

In the system there are two types of users. In the actor description they are listed and described.

* PC player: controls the car from PC keyboard but still must look at the console display to see the game.
* Console player: controls the car with joystick which is a part of the board.
  1. Use case description.

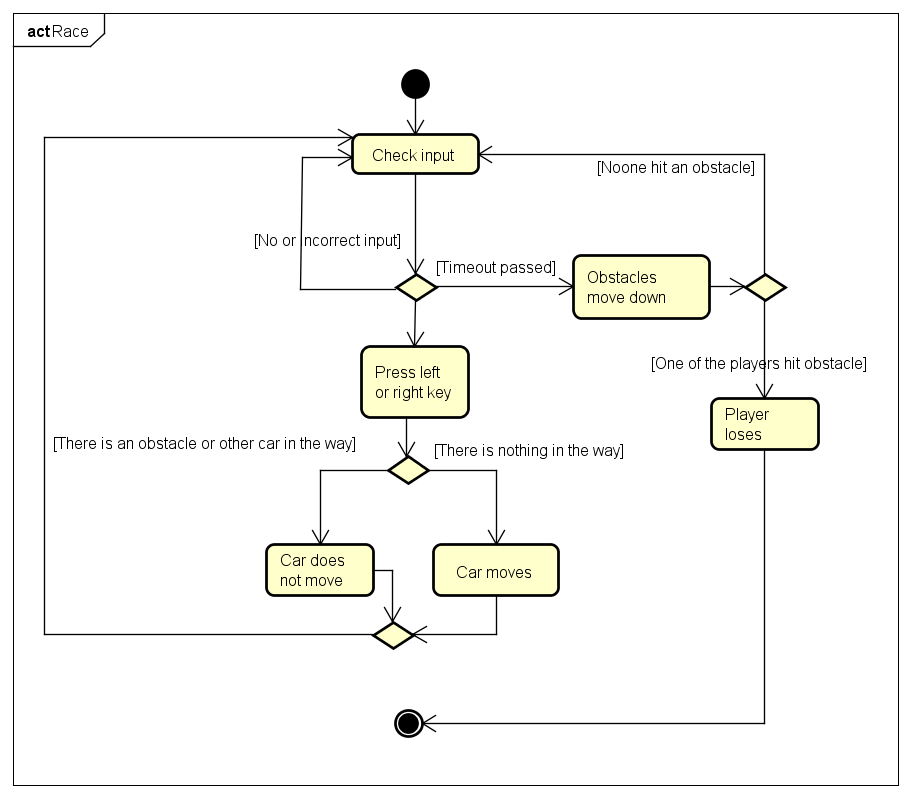
Here we show the use case description of the most essential action of the the game which is moving the car.



*Figure 2 – the use case description*

* 1. Activity diagram

The Race Activity Diagram displays how the car movement is managed. It is in essence the flow of the “Move car” use case associated with all the possible reactions from the system in terms of game logic. The system should constantly check for an input from a player as well as move the obstacles towards the cars. Once an input is registered, according to the desired direction of the movement, the system will either move the car if it is possible or leave it in the current position. If a car hits an obstacle in front, the player controlling the car is losing the round.



*Figure 3 – the Race Activity Diagram*

1. Design

The design phase is yet another planning stage, similar to the analysis. But this time, how the system can be implemented is discussed. Namely, here are specified the design choices that were made, as well as all the organization of the tasks including the assignment of periods and deadlines. It is important to note that this section provides the best overview over the core of the application.

* 1. The Tasks

The application that is being developed, intended to be real-time, will of course be composed of different tasks that will be responsible for implementing separate pieces of functionality. It is unavoidable that the tasks must communicate with each other and thus share resources, but in this section however, there will be a detailed description for all of the tasks present in the game console part of the application: Display updater task, Game logic task, Obstacles task, Joystick sampler task, Communication sender task, Communication receiver task.

3.1.1. Display updater task

The foundation the system is built on provides a sort of “API” or drivers that can be used to control the components of the game console. In this case, the display is a timer triggered ISR which updates the 14x10 LED with a constant rate from a predifined buffer. Basically, by changing the contents of the buffer, the display can be controlled. Because it is easier to work with 2-dimensional arrays in terms of implementing the game logic, it was decided that there is a need for a static matrix that can be converted into the buffer needed for the driver to update the display. This convertion is in fact the role of this task. It is approximated that for good update rate, a standard frequency of 24 updates/second will suffice (this is the same rate for switching images in a movie so that the movement seems smooth). If a calculus is performed, this tasks should idealy perform around every 40 ms. Additionally, it can be assumed that this tasks could have one of the lowest priority because both the high frequency and uncrutialness for the performance of the system.

3.1.2. Joystick sampler task

The game console can provide a player with a joystick that can be used to control one of the cars in the game. In order be able to access the input from the joystick, it is necessary to understand which pins for the MCU are responsible for communicating with this peripheral. This can be observed from the datasheets of both of them. In essence, each of the position of the joystick will map to a certain pin being set. In this context, the joytick sampler task is responsible for checking the variation of the responsible pins and provide the system with a translated version of what command must be implemented. In order to maintain a certain degree of modularitty of the system, it is important that this task perform only the above mentioned role. Therefore, there arises the problem of transmitting commands from the input to the system in some sort of way. The best suited solution that was found was to implement a FreeRTOS queue, that would contain the commands and which could be accesible from other tasks. Normally, it would be expected that the joystick is sampled around 10 times per second. Doing it too frequently would result in registering too many commands per press. But obviously once a second is too slow for sampling efficiently. Assuming that around 10 times per second would be optimal, this task should run once in every 100ms. The priority of this task could be medium, assuming that small delays could not become a performance drowback.

3.1.3. Game logic task

serves the purpose of updating the Matrix correctly based on the input queue but insures that executed inputs make sense and checks if cars and obstacles collide to decide the looser.

3.1.4. Obstacles task

3.1.5. Communication sender task

3.1.6. Communication receiver task

* 1. Resources
     1. Game Data

14x10 matrix representing the display. We decided to use this representation because it is easier to manipulate it and imagine what is on the display. This matrix must be converted into the one-dimensional array to be send to display, which may cost some time and processor power but it will make the implementation of the game easier.

* + 1. Frame\_buf

An array which is a source for displaying. This was designed and implemented by Ib Havn our supervisor.

* + 1. Input queue

A queue accessed by the game logic task, joystick sampler and console receiver. Joystick sampler and console receiver push into this queue so any input from joystick or keyboard ends up here. Game logic pops it for input to process.

* + 1. Sender queue

A queue accessed by the console sender and computer receiver. Acknowledgements, Not-Acknowledgements or messages about victory/loss are pushed into the queue. Then sent across.

* + 1. Acknowledgement queue

A queue accessed by the console receiver and console sender as well as computer receiver and computer sender. When the console receives a message, it puts acknowledgement or not-acknowledgment to the queue.

* + 1. Cars

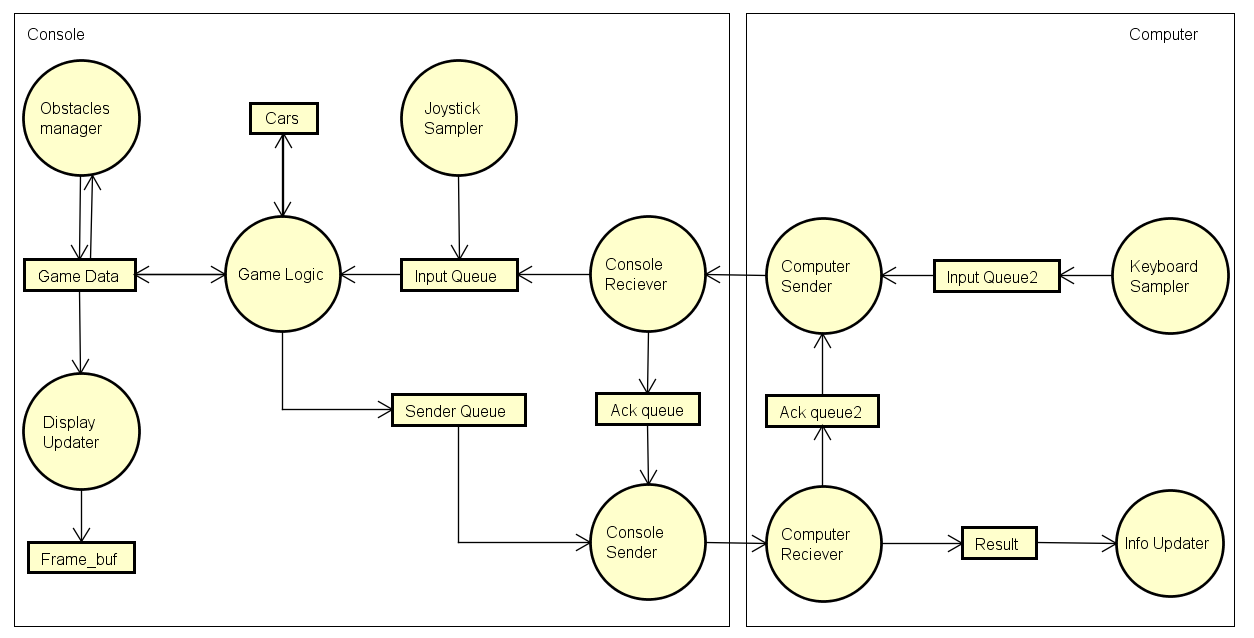
Two arrays of size two accessed by the game logic task. Each array representing one car. Every car has its own X and Y coordinate also represented in the matrix.

* 1. Resource management

Since we have quite a few shared resources which are not protected by freeRTOS we must use mutexes to protect them. Mainly the game data(matrix) which contains all important data. The matrix is modified by obstacles task and game logic therefore these two tasks will both try to take semaphore before maniulation of the game data then give it back once the procedure is executed.

* 1. Design diagram

For easier orientation and understanding all the tasks and the most essential resources are put into a diagram which shows how tasks communicate and read or manipulate resources.

To summarize the whole design, movement of the cars and their collision and constaints are maneged by Game Logic and outputted into Game Data together with obstacles which are generated and managed by Obstacles Manager. Movement of the cars is pushed into the Input Queue in the console by Joystick sampler which is checking joystick input and transforms it into a command readable by Game Logic. Similar think happens on the Keyboard sampler which takes input from keyboard on computer and pushes to Input Queue 2. Then we get to the communication which is handeled by a sender/reciever pair on each side. Computer Sender pops from Input Queue 2 encrypts the message through the protocal and sends it then it sets timer for an acknoledgement and keeps trying to pop it from Ack Queue 2. Console Reciever recieves the data and through the protocol translates them into the proper form and with a hamming code checks if everything is uncorrupted. Is so, it pushes an akcnowledgement into an Ack Queue and an input into the Input Queue. Console Sender pops an acknowledgement or a game result from the Ack Queue or the Sender Queue then use protocol to encrypt it and sends it across. Computer Reciever gets the data. If it is an acknowledgement it is push into the Ack Queue 2. Otherwise the acknowledgements is generated and pushed into Ack Queue 2 and also push the message into Result from which Info Updater updates the game statistics. This result comes from Game Logic when at least one of the cars hits an obstacle.

*Figure 4 – design diagram*

1. Implementation

In the design phase, we developed and implemented the components and application features. While we were working on the development, we also placed a lot of emphasis to the management of our time. Which means, we tried to optimize our schedules to work as effectively as we could and also at the end to have the best quality product. We met almost every day to coordinate the development of the program and did use the “git” software for code-sharing.

* 1. Database
     1. SQL Database

For this project we have used an SQL database in order to save the data about the users and the elections. To create a connection between the database and the program itself we have used a JDBC driver and created a DBManager class, which is responsible between logical communication between the database and the controller.

The SQL code:

CREATE SCHEMA Vote;

The Users table contains information about the users using the system. A name (username), a password, whether the user is an admin or not and whether the user (voter) has already voted or not.

CREATE TABLE Users (

name VARCHAR(40),

password VARCHAR(40),

admin BOOLEAN,

voted BOOLEAN);

The Candidates table contains information about the candidates of the actual election. A name, a position (name of the position) and a vote count (number of votes the candidate has).

CREATE TABLE Candidates(

name VARCHAR(40),

position VARCHAR(40),

voteCount INT NOT NULL);

The Election table contains information about the state of the actual election, whether if it’s active or not.

CREATE TABLE Election(

active BOOLEAN);

The Positions table contains information about the positions in the election (names of the positions).

CREATE TABLE Positions(

name VARCHAR(40));

The users (admins and voters) input data into the GUI. This data is handled by the controller, which it transfers to the DBManager after certain logical functions. Then the DBManager communicates with the SQL database using up the data it has recieved.

* + 1. Database driver

The **logIn** method is one of the most important methods in the **DBManager** class, it returns a User object (Voter or Admin) or null to the Controller, which it will use to determine what action to take next.

**public** User logIn(String username, String password)

{

Boolean adminBoolean = **false**; //It will store a boolean whether the searched user in the database is an admin or not.

Boolean voted = **true**; //It will store a boolean whether the voter in the database has already voted or not.

Boolean active = **false**; //It will store a boolean whether the election is active or not.

**try**

{

connection = DriverManager.*getConnection*(

"jdbc:postgresql://localhost:5432/postgres", "postgres", "pass");

}

**catch** (SQLException e)

{

}

// Returning data from the database.

**try**

{

PreparedStatement statement = connection.prepareStatement(

"SELECT \* FROM Users WHERE name = ? AND password = ?");

statement.setString(1, username);

statement.setString(2, password);

ResultSet result = statement.executeQuery();

**while** (result.next())

{

adminBoolean = result.getBoolean("admin");

voted = result.getBoolean("voted");

}

statement = connection.prepareStatement("SELECT \* FROM election");

result = statement.executeQuery();

result.next();

active = result.getBoolean("active");

}

**catch** (SQLException e)

{

}

**finally**

{

**try**

{

connection.close();

}

**catch** (SQLException e)

{

}

}

// Creating a User object based on the state of the boolean objects.

**try**

{

**if** (adminBoolean)

{

admin = **new** Admin(username, password);

**return** admin;

}

**else** **if** (!voted && active)

{

voter = **new** Voter(username, password, voted);

**return** voter;

}

}

**catch** (Exception e)

{

}

**return** **null**;

}

The **vote** method is mainly responsible of recording the votes of the voters in the database, incrementing the vote count of a certain candidate based on the name of the candidate in the method’s argument (which is provided by the controller).

**public** **void** vote(Candidate candidate)

{

**...**

**try**

{

PreparedStatement statement = connection.prepareStatement(

"UPDATE Candidates SET voteCount = ? WHERE name = ?");

candidate.giveVote();

statement.setInt(1, candidate.getVotes());

statement.setString(2, candidate.getName());

statement.executeUpdate();

statement = connection

.prepareStatement("UPDATE Users SET voted = 'true' WHERE name = ?");

statement.setString(1, voter.getName());

statement.executeUpdate();

}

The **reset** method makes it possible to empty the Candidates and the Positions table in the database, also making sure that voter users are allowed to vote again on a new election.

**public** **void** reset()

{

...

**try**

{

PreparedStatement statement1 = connection

.prepareStatement("DELETE FROM Candidates");

PreparedStatement statement2 = connection

.prepareStatement("DELETE FROM Positions");

PreparedStatement statement3 = connection

.prepareStatement("UPDATE Users SET voted = 'false' WHERE voted = 'true'");

statement1.executeUpdate();

statement2.executeUpdate();

statement3.executeUpdate();

}

* 1. Controller

**Log in** method ask the database for the list of the users with name and password. Then it compares this list with the username and password given in the login field. After it, when it finds the user, which owns that username and password, it will open a new window depending on wat kind of user he is. If admin, then admin window, if voter then voter window.

The code is shown below.

**public** **void** logIn(String name, **char**[] passwordIn)

/\*

\* Check if the password and name are in the database by calling the database

\* opening the admin window if its a admin

\* opening the voter window if its a voter

\*/

{

**if** (!name.equals(""))

{

String password = **new** String(passwordIn);

user = **new** User(name, password);

**try**

{

**if** (dbm.logIn(name, password) **instanceof** Admin)

{

logIn.close();

adminW = **new** AdminWindow(**this**);

logIn.clearFields();

}

**else** **if** (dbm.logIn(name, password) **instanceof** Voter)

{

logIn.close();

voterW = **new** VoterWindow(**this**);

logIn.clearFields();

}

**else**

{

logIn.clearFields();

JOptionPane.*showMessageDialog*(**new** JFrame(), "Unable to log in. Check your username and password. You can only vote once.");

//The option pane shows the dialog message if something is wrong with the password or username

}

}

**catch** (Exception e)

{

}

}

}

**Vote** method is in a close relationship with another methods called **addVote** and **okPressed**. The **addVote** method is collecting the votes which have been selected by the voter in the GUI drop down. When the user clicks the ok button the **okPressed** method is being called. This method is just calling the vote method.

Vote method is shown below

**public** **void** vote()

{

**for** (**int** i = 0; i < voteList.size(); i++)

{

**try**

{

dbm.vote(voteList.get(i));

}

**catch** (Exception e)

{

}

}

voteList = **new** ArrayList<Candidate>();

}

**addVote** method is shown below

**public** **void** addVote(String name, String pos)

{

**for** (**int** i = 0; i < voteList.size(); i++)

{

**if** (voteList.get(i).getPosition().equals(pos))

voteList.remove(i);

}

**try**

{

voteList.add(dbm.getCandidate(name, pos));

}

**catch** (Exception e)

{

}

}

1. Testing

The program has been tested by the using of all the required features through the GUI with multiple clients running at one time.

|  |  |
| --- | --- |
| * Log in as an administrator with his username and password | Passed |
| * Log out | Passed |
| * Change password | Passed |
| * Create position | Passed |
| * Delete positions together with all the candidates for that position | Passed |
| * Add candidate after selecting a position | Passed |
| * Remove candidates | Passed |
| * Start or stop election | Passed |
| * Remove everything from the tables of positions and candidates while allowing all the voters in the database to vote again | Passed |
| * See the results of the election at any time | Passed |
| * See all the registered candidates and positions | Passed |
| * Log in as a voter with his username and password if he did not vote yet and if the election has been started by an admin account | Passed |
| * Select multiple positions and one candidate for every position to vote for | Passed |
| * Send his votes to the database | Passed |
| * Log out without voting | Passed |
| * Preventing voters from voting more than once | Passed |

1. Discussion

There are few design choices that should be discussed more in-deth.

* 1. AdminWindow

We decided to use JTabbedPane on the AdminWindow to avoid having to many windows opening and closing and having buttons to switch between them at the same time. We were also interested about learning how to use the JTabbedPane.

* 1. RMI

The reason for using RMI for client-server implementation of the program is that we do not have to write any protocols or anything else. We just get the remote interface to control DBManager on the server so the Controller of the Client can retrieve and save data.

* 1. Stop/start election

We decided on having possibility do stop election because it seems to be right to not be able to vote outside the election period. And also to disable administrator from inserting new candidates or positions when the election is running.

* 1. No history of votes

We decided to not keep any track or history of votes to keep any election running on our system anonymous.

1. Results

After the code was done and put together it was time to try to run it on an RMI server and RMI client on two different computers and test how it works. We created a small table of users just for testing purposes. After assuring the key features works we tested the less important ones and made sure they work as well. During the testing we realized we forgot about few features that could be useful and clever to have. So we returned to the code a few times before calling it fully functional. When we were done we decided to polish it, delete the fields we did not use in the end, unnecessary methods and code that was commented out.

The software is capable of creating an environment for running an elections. It offers an anonymous election for multiple position with multiple candidates. The database of users has to be imported through another program. It prevents voters from voting more than once and it is possible to prevent voters from voting by stopping the election through the administrator. All the values except registered voters can be deleted by resetting the system.

1. Conclusion

The project was about creating a client-server system which uses a database on the topic of our choice. We choose to implements a system for DSR elections. The system was developed according to the requirements set at the beginning of the project and the result is relatively generic election system that can be used even for different types of elections. It is even possible to run multiple clients at a time while keeping the functionality. The table of users has to be imported externally and cannot be managed through the system. The table of positions and the table of candidates can managed through the system and can be also imported.

1. References

**[1]** Gaddis, G.T. (2014). Creating a simple GUI app, Starting Out with Java. Great Britain: Pearson Education.

**[2]** Gaddis, G.T. (2014). Handling events, Starting Out with Java. Great Britain: Pearson Education.

**[3]** <https://www.youtube.com/watch?v=Ou2A-JWszVA> December 2015

**[4]** http://mathbits.com/MathBits/Java/arrays/ABCSort.htm December 2015

**[5]** <http://www.w3schools.com/xml/> December 2015

**[6]** https://docs.oracle.com/javase/tutorial/uiswing/layout/gridbag.htmlDecember 2015

**[7]** <http://stackoverflow.com/questions/5212983/jlist-add-remove-item> December 2015