VIA UNIVERSITY COLLEGE
ICT ENGINEERING

Project Report

Betting system

Group 10, SEP2Z-S16
Atanas Latinov (239 841)
Daniel Hamarik (239 857)
Stefan Alexiev (240 278)

Supervisors
Henrik Kronborg Pedersen
Jan Munch Pedersen

May 2016

SEPI2-S16, Class Z, Group 10

Table of Contents

List of figures and tables

1. Abstract

With the increasing number of sport events of various kind, both on a national and international level, and the huge and still growing spectator base of those events, it is only logical to use the power of today's technology and the internet to provide a system that enables people to bet on events on the other side of the planet in the comfort of their own home. For this reason, we have created a smart betting solution whose aim is to address those problems. The betting system is built with a client-server architecture (RMI) with a fully functional PostgreSQL database. All data is stored digitally in the database and it is safe to say that working with it is much faster and the user's productivity is increased tremendously. With this new and powerful system, it is possible for users to register and immediately place bets on different kinds of events.

The program is storing all of the users' data such as profile information, balance and user's betting history to monitor his/her progress and profit. The program has a simple, user friendly interface which provides quick access to common features and commands.

The latest tests of the program have proven that it is a fully operational software product that satisfies all the requirements and demands and show that the program is capable of facilitating bet processing and the procedure of determining the outcome for the users, depending on their choices.

2. Introduction

Sports' betting has been a popular pastime for many throughout history, and there are probably more people than ever that enjoy it these days. It has been around for a while now but it continues to evolve. Nowadays, with the emerging of new events, many bookmakers fail to cover many of them that are for example very popular in other parts of the world. Not only that but also many people do not have the chance to place their bets because they are either far from a bookmaker or it is just impossible for them. It is clear that these complications and many more can be easily avoided just by using the capabilities of today's technology.

The aim of this project is to create a smart betting solution that enables its users to easily place bets, review match scores and see their betting history. Placing bets on sporting events has never been easier than it is today, thanks to the power of the internet. Online betting allows users to have access to a full list of betting events from their desktop.

This report presents the technical details for a "Betting system" application. The application attempts to answer all the problems mentioned above. The betting system offers a very simple and intuitive user interface, which is focused on the main functionalities, the program is fully-featured to be useful in everyday scenarios. This project report discusses functional, non-functional requirements, main features, overall structure of the system and the user interface design.

3. Requirements

3.1 Functional requirements

This section describes all functional (behavioral) requirements for the system. The following prioritized list of requirements details typical scenarios of using the system. The list is based on the use case model and all requirements are user and system facing.

I. User can place bets on multiple events
II. User is able to retrieve his/her bets on demand
III. User can see a daily offer of matches with a detailed description
IV. User can filter events based on their types and a time frame
V. Administrator can insert or edit matches
VI. Administrator is able to set results for a selected match
VII. Winnings are distributed according to match results
VIII. Users have to log in in the system with their password
IX. User is able to create a new account with a permanent username
X. User can edit his/her profile
XI. Administrator is able to add/remove funds from users' accounts

3.2 Non-functional requirements

In this section, we list the most important technically-oriented attributes. Non-functional (quality) requirements do not describe what the software will do, but how the software will do it. The following list shows the non-functional requirements, which are based on the project specifications.

I. System must work over a network
II. System must use RMI or socket programming
III. The backend of the system must be database based
IV. Functional dependency of the database
V. System must implement design patterns
VI. System must be scalable
VII. System must be capable of updating the contents aimed at the users
VIII. System must implement authentication as a form of security

4. Analysis

4.1 Use case modeling

We have created a model of the system's functionality and environment during the first phase of unified process – Inception phase. The use case model reflects typical scenarios when using the betting system. The following section discusses the use case diagram, description and activity diagrams. All remaining documentation created during analysis of the system can be found in the Appendix.

4.1.1 Use case diagram

The following use case diagram provides a succinct visual context of the external actors and how they interact with the system. In our case, we have two different actors: User and Administrator. We consider the use case diagram as an excellent representation of the system context.

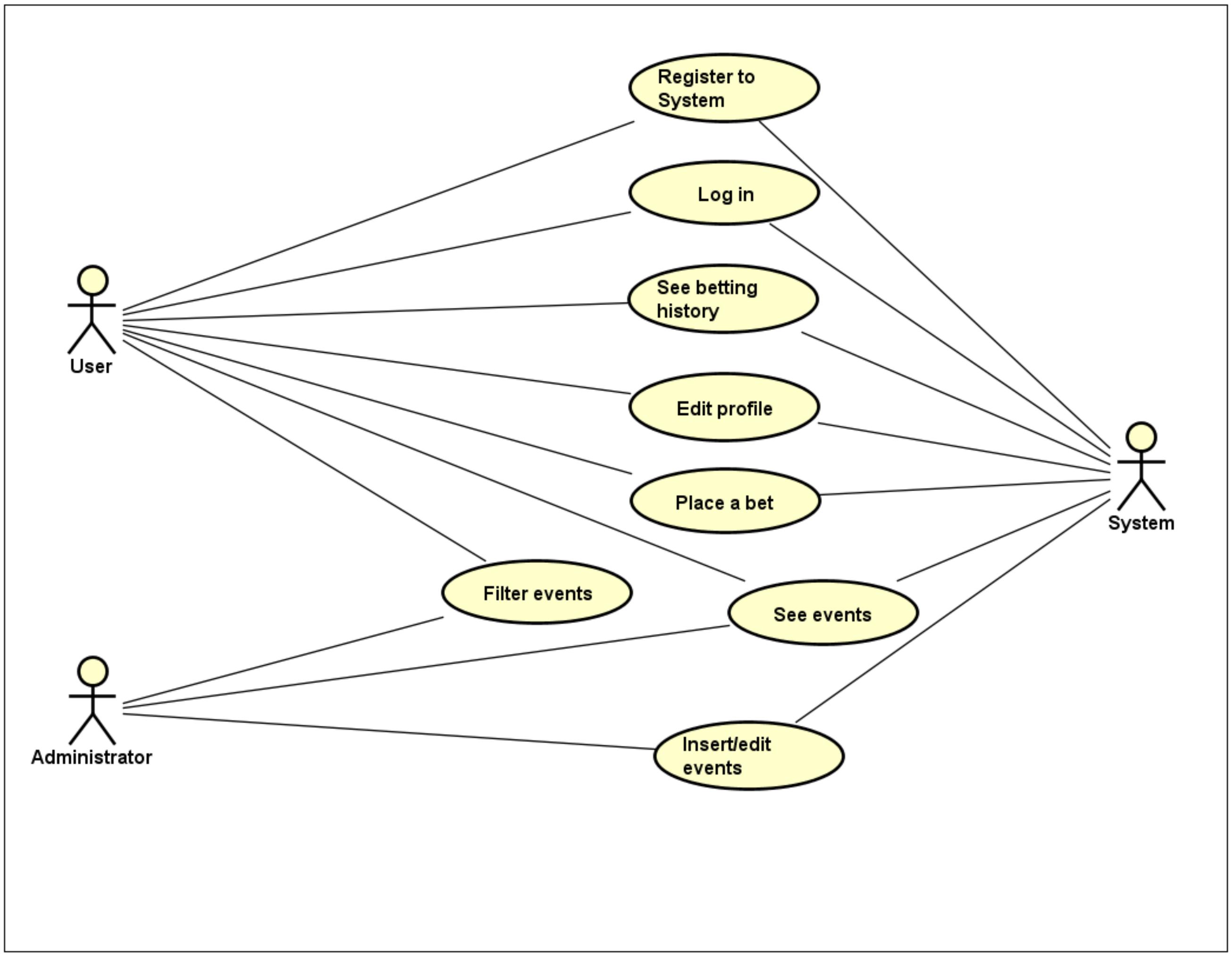
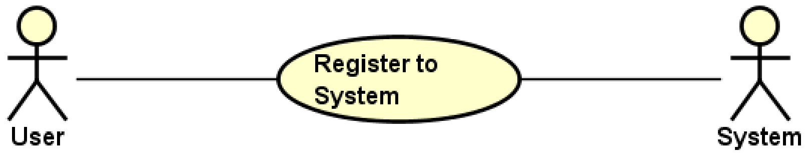
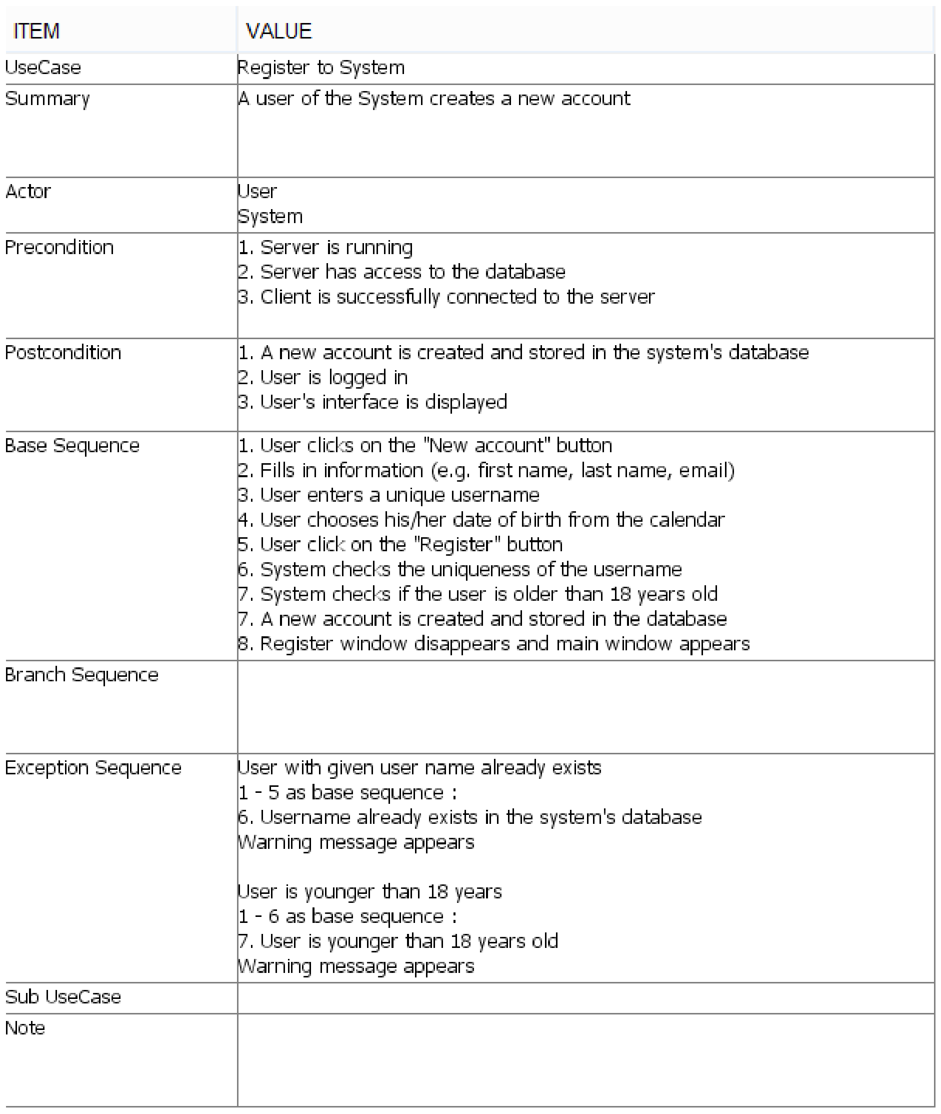


Figure 1. Use case diagram

4.1.2 Use case description

During the second phase of unified process, we have created a detailed description of every use case model. A use case description about the "Register to system" action is provided here to serve as an example.





ITEM	VALUE
UseCase	Register to System
Summary	A user of the System creates a new account
Actor	User System
Precondition	1. Server is running 2. Server has access to the database 3. Client is successfully connected to the server
Postcondition	1. A new account is created and stored in the system's database 2. User is logged in 3. User's interface is displayed
Base Sequence	1. User clicks on the "New account" button 2. Fills in information (e.g. first name, last name, email) 3. User enters a unique username 4. User chooses his/her date of birth from the calendar 5. User click on the "Register" button 6. System checks the uniqueness of the username 7. System checks if the user is older than 18 years old 7. A new account is created and stored in the database 8. Register window disappears and main window appears
Branch Sequence	
Exception Sequence	User with given user name already exists 1 - 5 as base sequence : 6. Username already exists in the system's database Warning message appears User is younger than 18 years 1 - 6 as base sequence : 7. User is younger than 18 years old Warning message appears
Sub UseCase	
Note	

Figure 2. Register in the System – use case description

4.2 Activity diagram

This activity diagram describes the dynamic aspect of placing a bet in the system. It is a flow chart which represents the user, who is choosing different matches and at the end is placing a bet which is stored in the system's database for later evaluation. All remaining activity diagrams are in the Appendix.

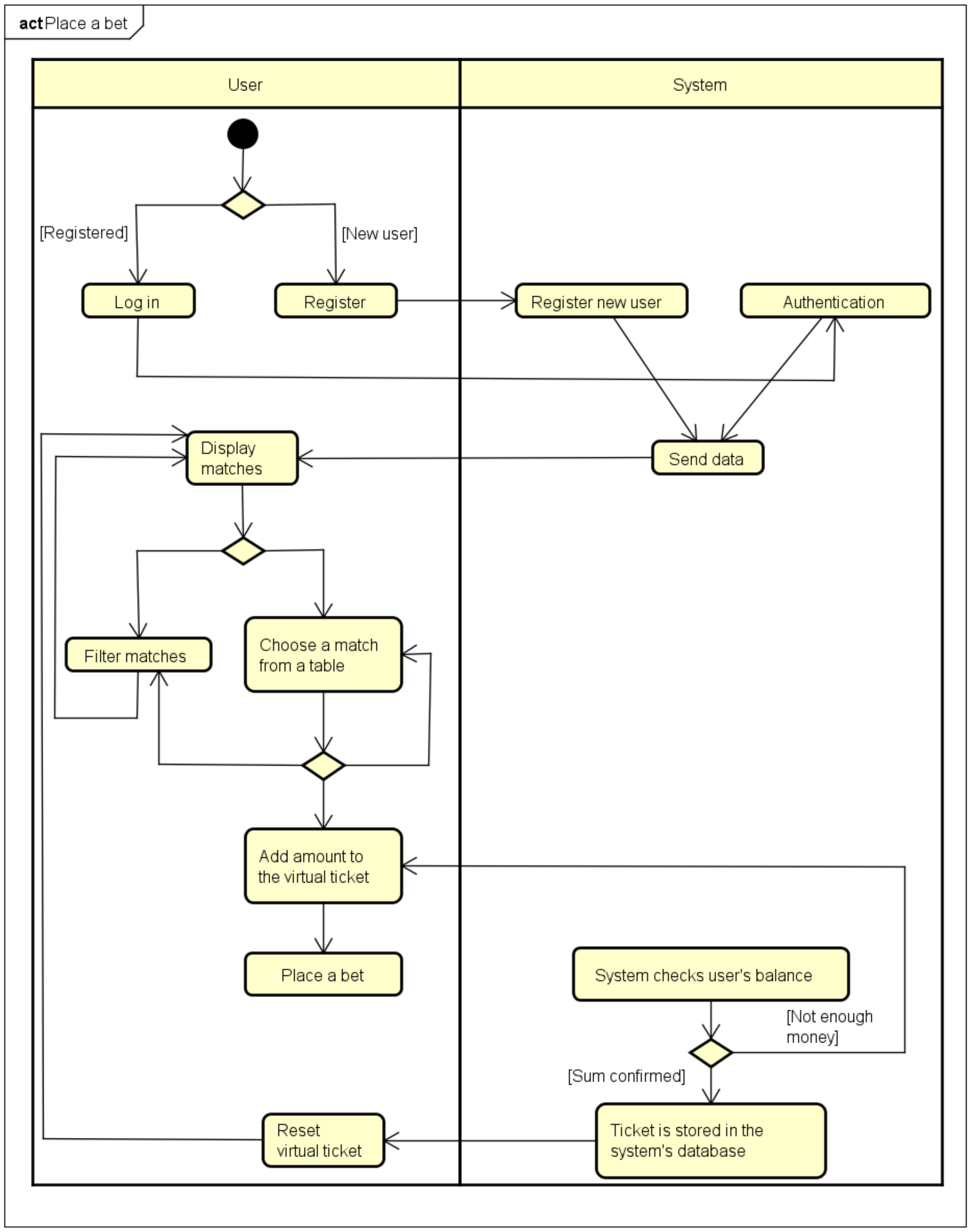


Figure 3. Place a bet – activity diagram

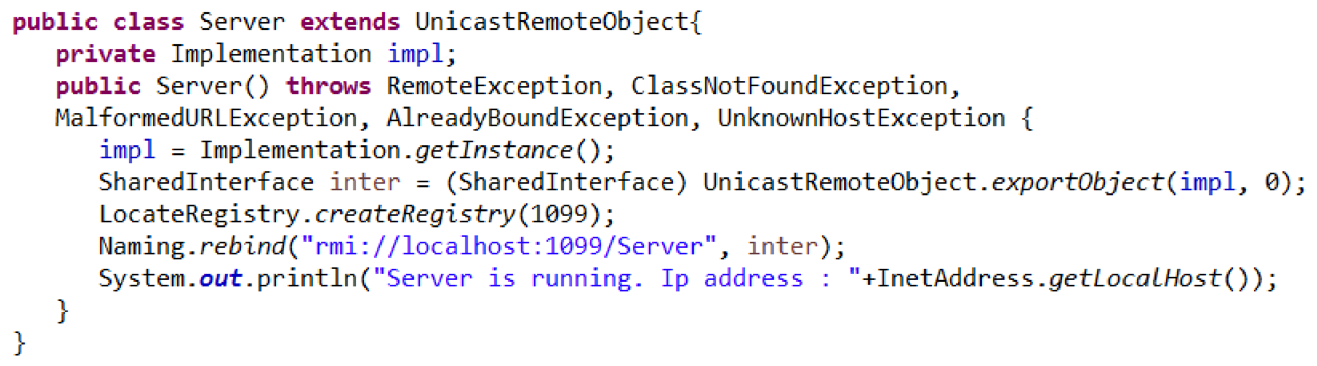
5. Client-server model

5.1 Java Remote Method Invocation

The client-server connection is based on a communication over a computer network on separate hardware. In this system, the server host runs one program which rebinds the shared interface to all clients. A client does not share any of its resources, but requests a server's contents and methods from the shared interface. Clients therefore initiate communication sessions with servers, which await incoming requests. The Client-server communication is provided by RMI (Remote Method Invocation), which allows an object, running in one Java virtual machine to invoke methods on an object running in another Java virtual machine. We find RMI as a very flexible kind of a Remote Procedure and using it is hugely beneficial in this case since the interaction with the database is done on the server side. The Client does not need a direct access to the database; he needs to call methods for inserting and reading from the database. All tasks are executed on the server side and results are sent over the network to the client. It means that the communication with the database is completely separated from the client.

Server:

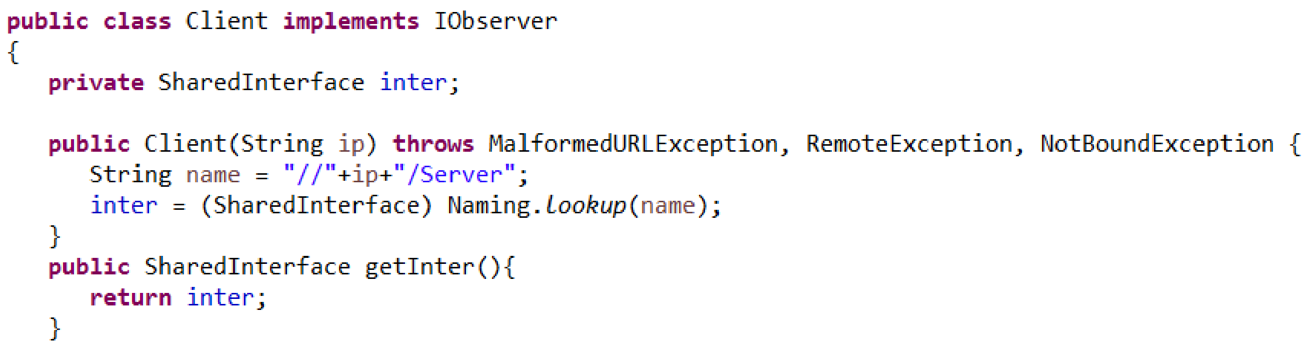
The following code snippets show the Server class, which is responsible for overall communication with clients. As is presented below, the server class contains an instance of the Implementation class, which is responsible for all methods called by the clients. This instance is casted to SharedInterface and rebound to all clients on the port 1099 from its IP address.



```
public class Server extends UnicastRemoteObject{
   private Implementation impl;
   public Server() throws RemoteException, ClassNotFoundException,
   MalformedURLException, AlreadyBoundException, UnknownHostException {
      impl = Implementation.getInstance();
      SharedInterface inter = (SharedInterface) UnicastRemoteObject.exportObject(impl, 0);
      LocateRegistry.createRegistry(1099);
      Naming.rebind("rmi://localhost:1099/Server", inter);
      System.out.println("Server is running. Ip address : "+InetAddress.getLocalHost());
   }
}
```

Figure 4. Server

Client:
The Client is looking for the SharedInterface on the passed IP address as an argument for the method. If the lookup is successful, the client can call all methods from the interface which are executed on the server side and results are sent back to the client.



```java
public class Client implements IObserver
{
   private SharedInterface inter;

   public Client(String ip) throws MalformedURLException, RemoteException, NotBoundException {
      String name = "//"+ip+"/Server";
      inter = (SharedInterface) Naming.lookup(name);
   }
   public SharedInterface getInter(){
      return inter;
   }
```

Figure 5. Client

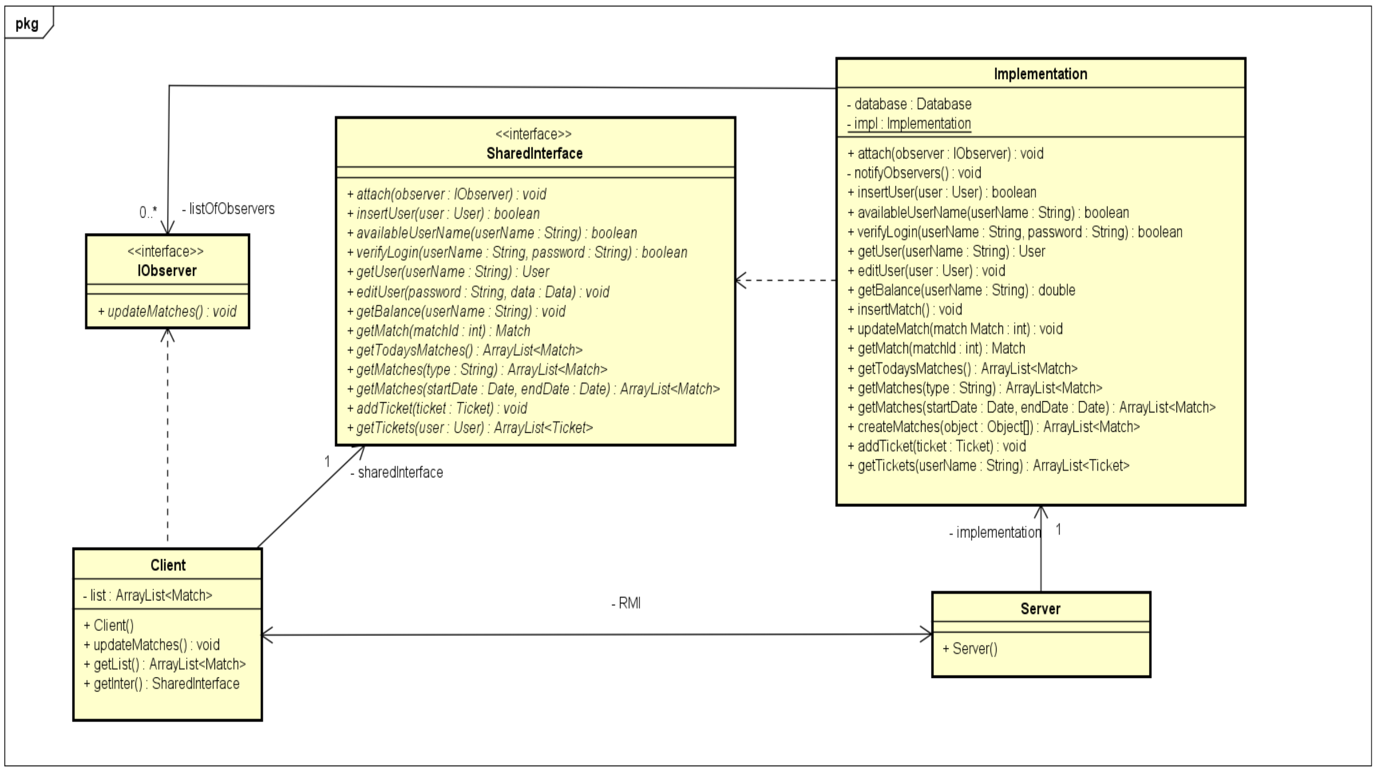
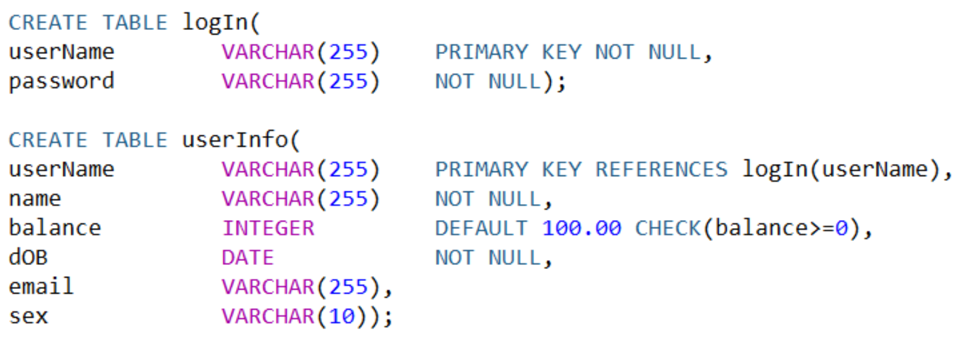

pkg
Implementation
- database : Database
- impl : Implementation
+ attach(observer : IObserver) : void
- notifyObservers() : void
+ insertUser(user : User) : boolean
+ availableUserName(userName : String) : boolean
+ verifyLogin(userName : String, password : String) : boolean
+ getUser(userName : String) : User
+ editUser(user : User) : void
+ getBalance(userName : String) : double
+ insertMatch() : void
+ updateMatch(match Match : int) : void
+ getMatch(matchId : int) : Match
+ getTodaysMatches() : ArrayList<Match>
+ getMatches(type : String) : ArrayList<Match>
+ getMatches(startDate : Date, endDate : Date) : ArrayList<Match>
+ createMatches(object : Object[]) : ArrayList<Match>
+ addTicket(ticket : Ticket) : void
+ getTickets(userName : String) : ArrayList<Ticket>
<<interface>>
SharedInterface
+ attach(observer : IObserver) : void
+ insertUser(user : User) : boolean
+ availableUserName(userName : String) : boolean
+ verifyLogin(userName : String, password : String) : boolean
+ getUser(userName : String) : User
+ editUser(password : String, data : Data) : void
+ getBalance(userName : String) : void
+ getMatch(matchId : int) : Match
+ getTodaysMatches() : ArrayList<Match>
+ getMatches(type : String) : ArrayList<Match>
+ getMatches(startDate : Date, endDate : Date) : ArrayList<Match>
+ addTicket(ticket : Ticket) : void
+ getTickets(user : User) : ArrayList<Ticket>
0..*
- listOfObservers
<<interface>>
IObserver
+ updateMatches() : void
1
- sharedInterface
Client
- list : ArrayList<Match>
+ Client()
+ updateMatches() : void
+ getList() : ArrayList<Match>
+ getInter() : SharedInterface
- RMI
- implementation
1
Server
+ Server()

Figure 6. Client-server architecture

6. Database

6.1 Database design

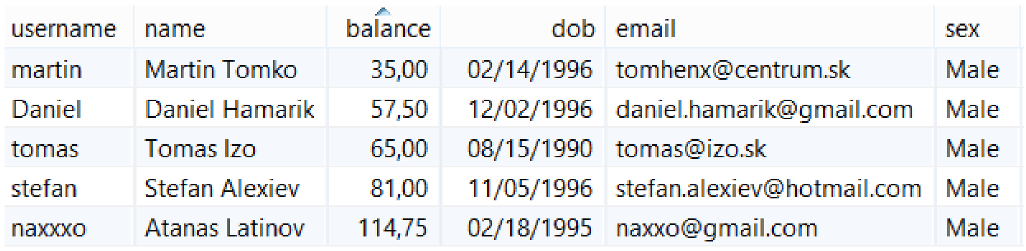
One of the requirements was that the backend of the system must be database based, so we decided to use a PostgreSQL database. This relational database is the heart of our system because everything is stored in it. The database architecture has been developed from scratch, followed by the usage of normalization design techniques. The main objective of normalization was to create an accurate representation of the data, relationships between the data, and constraints on the data that is pertinent in the betting system. We went through a series of tests (normal forms) during the development of the database and it resulted in a fully functional database, which consists of 5 dependent tables and one trigger function. The interested reader can check the whole process of normalization, described in the Appendix.



```
CREATE TABLE logIn(
userName        VARCHAR(255)    PRIMARY KEY NOT NULL,
password        VARCHAR(255)    NOT NULL);

CREATE TABLE userInfo(
userName        VARCHAR(255)    PRIMARY KEY REFERENCES logIn(userName),
name            VARCHAR(255)    NOT NULL,
balance         INTEGER         DEFAULT 100.00 CHECK(balance>=0),
dOB             DATE            NOT NULL,
email           VARCHAR(255),
sex             VARCHAR(10));
```

Figure 7. Table examples

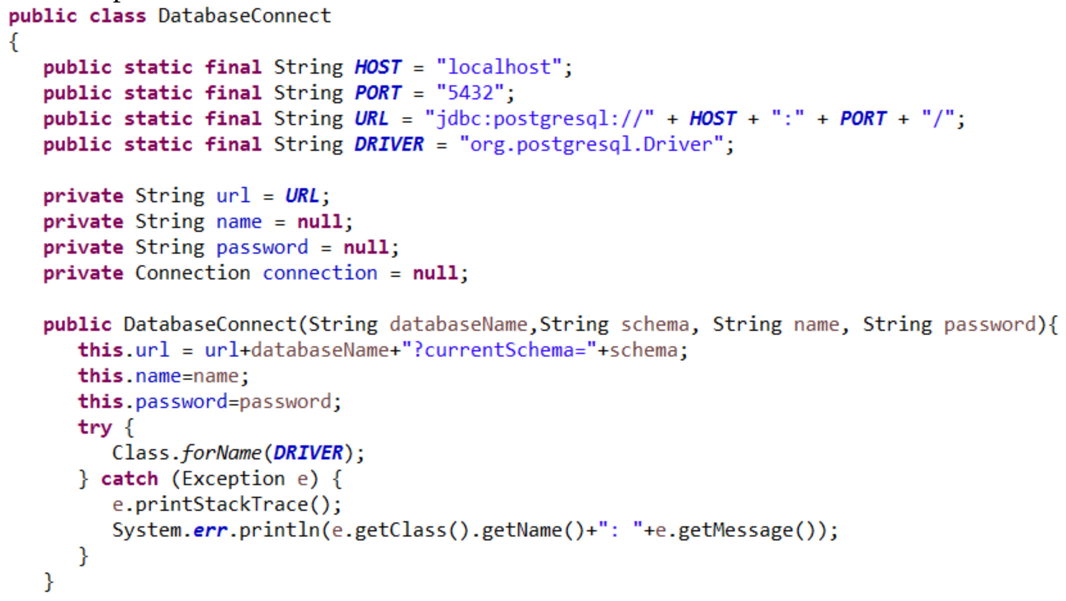


username	name	balance	dob	email	sex
martin	Martin Tomko	35,00	02/14/1996	tomhenx@centrum.sk	Male
Daniel	Daniel Hamarik	57,50	12/02/1996	daniel.hamarik@gmail.com	Male
tomas	Tomas Izo	65,00	08/15/1990	tomas@izo.sk	Male
stefan	Stefan Alexiev	81,00	11/05/1996	stefan.alexiev@hotmail.com	Male
naxxxo	Atanas Latinov	114,75	02/18/1995	naxxo@gmail.com	Male

Figure 8. userInfo table

6.2 Java database connection

This section describes the connection between the database and our system written on the Java platform. In the system, database access occurs on the server side and the GUI processing with all data occurs on the client side. JDBC provides easy access to the database and RMI distributes the objects that make up the whole application. JDBC allows constructing SQL statements in Java and executing them in the database with results returned as Java objects. Thanks to this type of connection, we do not have to worry about how it is routing SQL statements, hence database manipulation is straightforward. The following code snippets show how Java connects to the database. In the second snippet is an example of a prepared statement, which in combination with Varargs, creates a very powerful tool for database data manipulation.

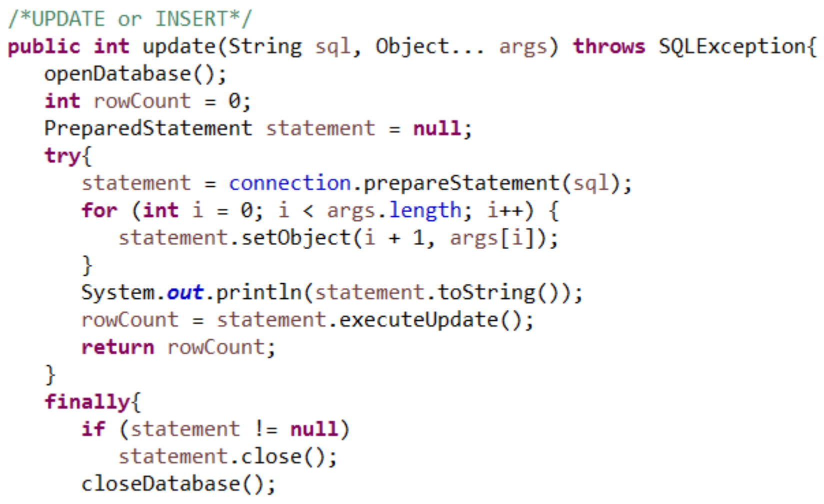


```java
public class DatabaseConnect
{
   public static final String HOST = "localhost";
   public static final String PORT = "5432";
   public static final String URL = "jdbc:postgresql://" + HOST + ":" + PORT + "/";
   public static final String DRIVER = "org.postgresql.Driver";

   private String url = URL;
   private String name = null;
   private String password = null;
   private Connection connection = null;

   public DatabaseConnect(String databaseName,String schema, String name, String password){
      this.url = url+databaseName+"?currentSchema="+schema;
      this.name=name;
      this.password=password;
      try {
         Class.forName(DRIVER);
      } catch (Exception e) {
         e.printStackTrace();
         System.err.println(e.getClass().getName()+": "+e.getMessage());
      }
   }
```

Figure 9. Database connection



```java
/*UPDATE or INSERT*/
public int update(String sql, Object... args) throws SQLException{
   openDatabase();
   int rowCount = 0;
   PreparedStatement statement = null;
   try{
      statement = connection.prepareStatement(sql);
      for (int i = 0; i < args.length; i++) {
         statement.setObject(i + 1, args[i]);
      }
      System.out.println(statement.toString());
      rowCount = statement.executeUpdate();
      return rowCount;
   }
   finally{
      if (statement != null)
         statement.close();
      closeDatabase();
```

Figure 10. Prepared statement

7. Design patterns

7.1 Singleton design pattern

The implementation class is the core of this system because it implements the shared interface. It operates with unique data, which is loaded and saved from the system's database. Besides, methods from the Implementation class may need to be called from various places in the code. One solution is to pass an Implementation instance around as a parameter to all classes, which operate with its methods. This is possible but inconvenient so in this situation we find the singleton design pattern as very useful. With the singleton design pattern, there is always exactly one instance of a class allowed. The Implementation class has a private constructor and a static getInstance() method. With this approach, we have global visibility to this single instance, via the static getInstance method.

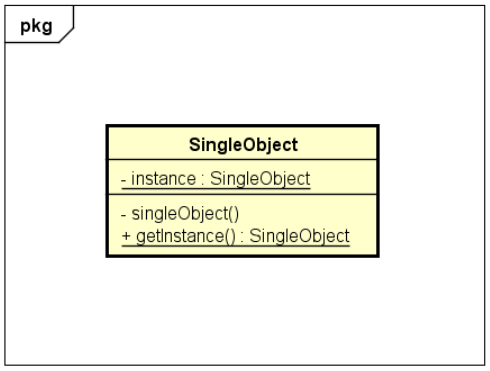
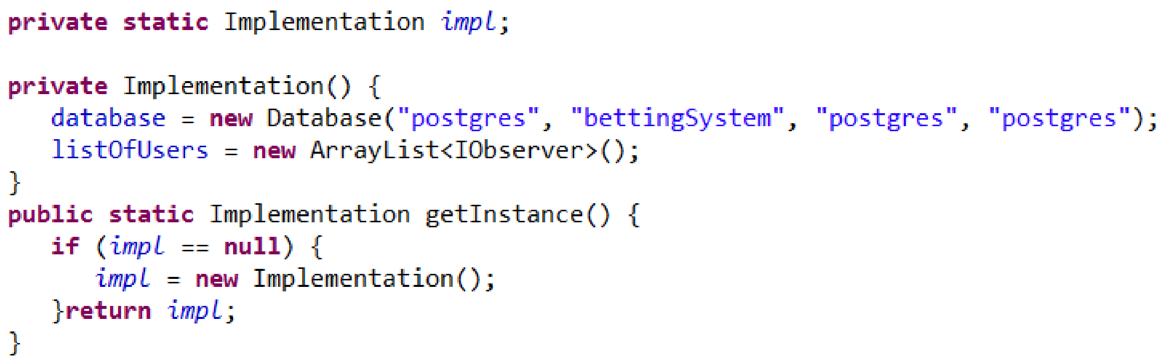


Figure 11. Singleton design pattern



```
private static Implementation impl;

private Implementation() {
   database = new Database("postgres", "bettingSystem", "postgres", "postgres");
   listOfUsers = new ArrayList<IObserver>();
}
public static Implementation getInstance() {
   if (impl == null) {
      impl = new Implementation();
   }return impl;
}
```

Figure 12. Singleton design pattern – code

7.2 Observer pattern

One of the most important requirements for a system such as this is that all clients are provided with actual event information at all times. This aspect is achieved by applying the observer design pattern to the code structure. The Observer pattern is also an integral part in the model–view–controller (MVC) architectural pattern, which is described in the next subsection. The Observer represents a one-to-many relationship so that when server inserts/edits an event, all clients are notified and updated automatically. The following UML diagram displays the Observer design pattern structure, used in this system.

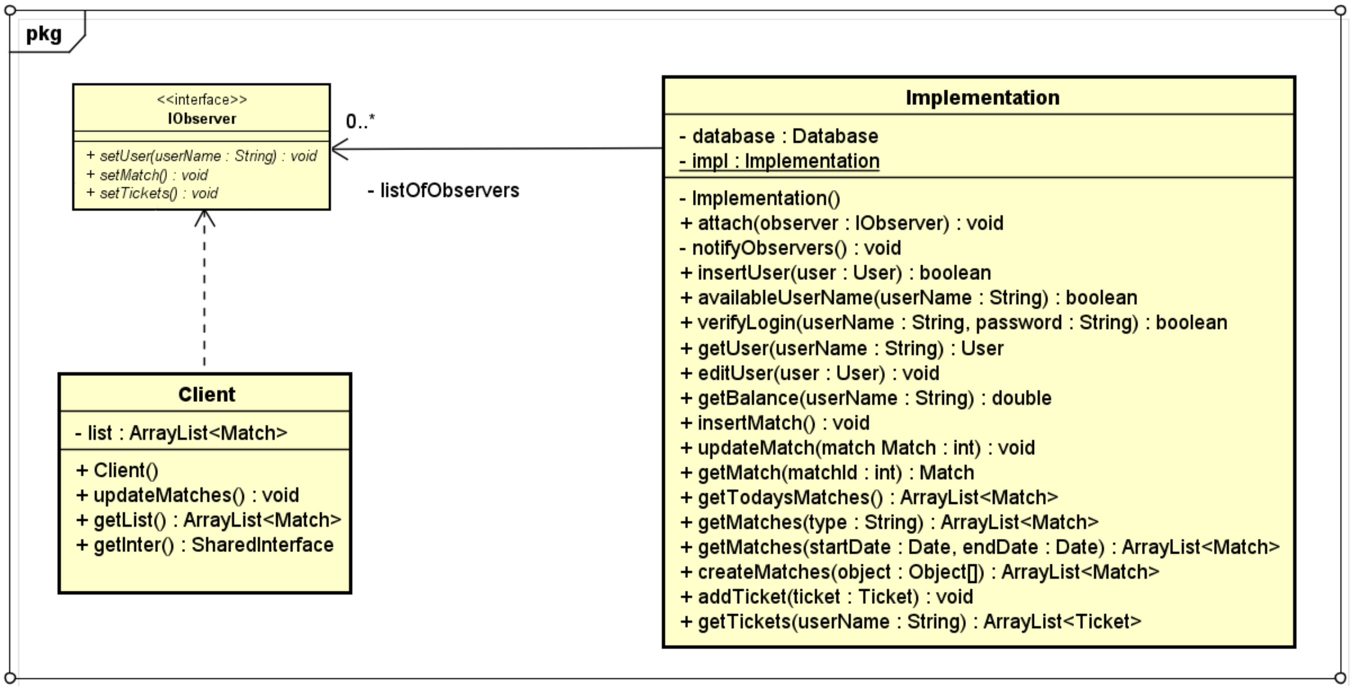

pkg
<<interface>>
IObserver
+ setUser(userName : String) : void
+ setMatch() : void
+ setTickets() : void
0..*
- listOfObservers
Implementation
- database : Database
- impl : Implementation
- Implementation()
+ attach(observer : IObserver) : void
- notifyObservers() : void
+ insertUser(user : User) : boolean
+ availableUserName(userName : String) : boolean
+ verifyLogin(userName : String, password : String) : boolean
+ getUser(userName : String) : User
+ editUser(user : User) : void
+ getBalance(userName : String) : double
+ insertMatch() : void
+ updateMatch(match Match : int) : void
+ getMatch(matchId : int) : Match
+ getTodaysMatches() : ArrayList<Match>
+ getMatches(type : String) : ArrayList<Match>
+ getMatches(startDate : Date, endDate : Date) : ArrayList<Match>
+ createMatches(object : Object[]) : ArrayList<Match>
+ addTicket(ticket : Ticket) : void
+ getTickets(userName : String) : ArrayList<Ticket>
Client
- list : ArrayList<Match>
+ Client()
+ updateMatches() : void
+ getList() : ArrayList<Match>
+ getInter() : SharedInterface

Figure 13. Observer design pattern

Subsequent snippets of the code show the main principles of the observer pattern. If the client is successfully connected to the server via RMI, he is also automatically associated with the list of observers. It means whenever a server adds a new match, all attached observers are notified and provided with a list of today's matches. As a result, all observers always have the actual list of matches they can bet on.

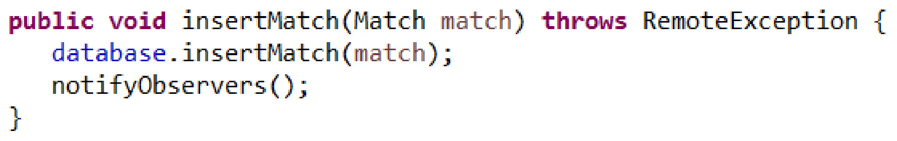
```
public Client(String ip) throws MalformedURLException, RemoteException, NotBoundException {
    String name = "//" + ip + "/Server";
    inter = (SharedInterface) Naming.lookup(name);
    todayList = new ArrayList<Match>();
    inter.attach(this);
}
```

Figure 14. Client's constructor

```
public void attach(IObserver observer) throws RemoteException {
    listOfUsers.add(observer);
}

public void notifyObservers() throws RemoteException {
    for (int i = 0; i < listOfUsers.size(); i++) {
        listOfUsers.get(i).setMatches();
    }
```

Figure 15. Server's methods

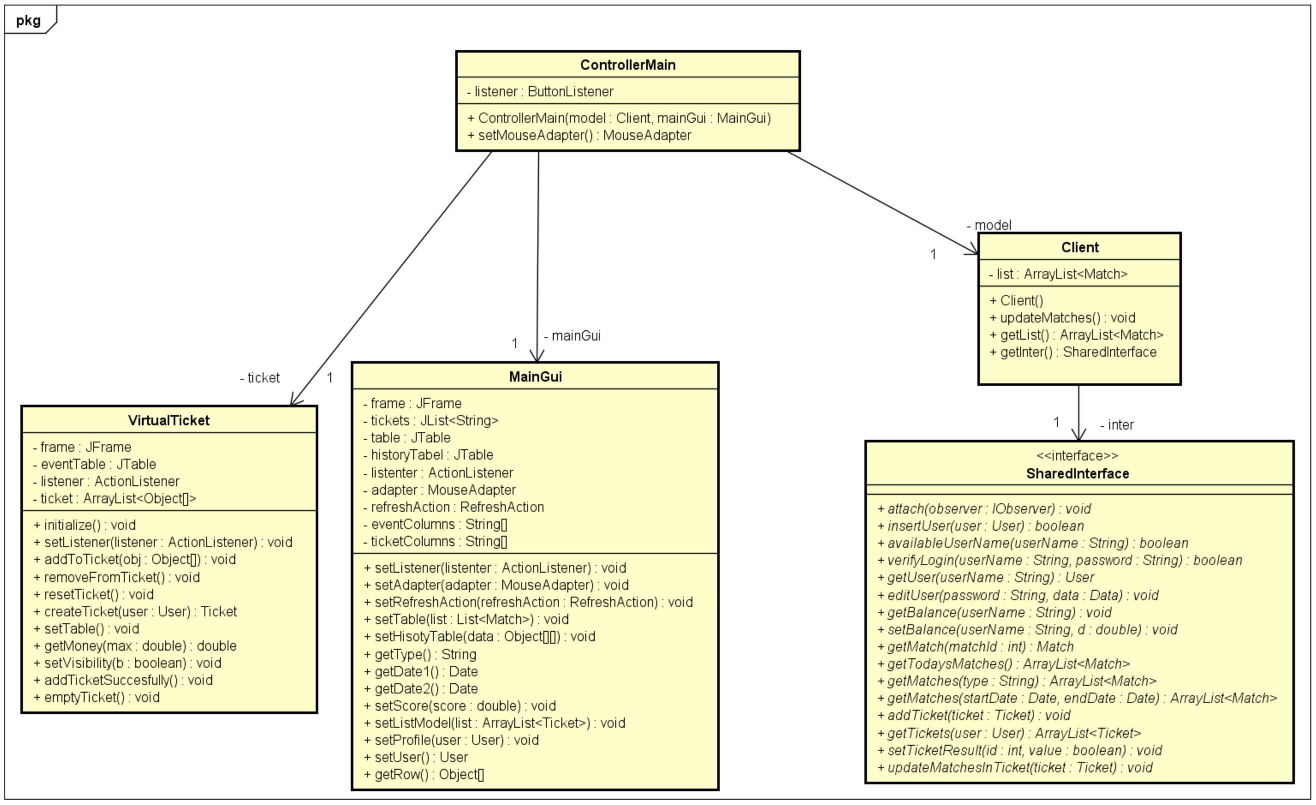


```
public void insertMatch(Match match) throws RemoteException {
   database.insertMatch(match);
   notifyObservers();
}
```

Figure 16. Match insertion

7.3 Model-view-controller

The client is based on the MVC design pattern that decomposes an application into three kinds of components. MVC allows the separation of the application's concerns. It divides a given software application into three interconnected parts, to separate internal representations of information from the ways that information is presented to or accepted from the user. The separation principle states that model objects should not have direct knowledge of the UI objects.



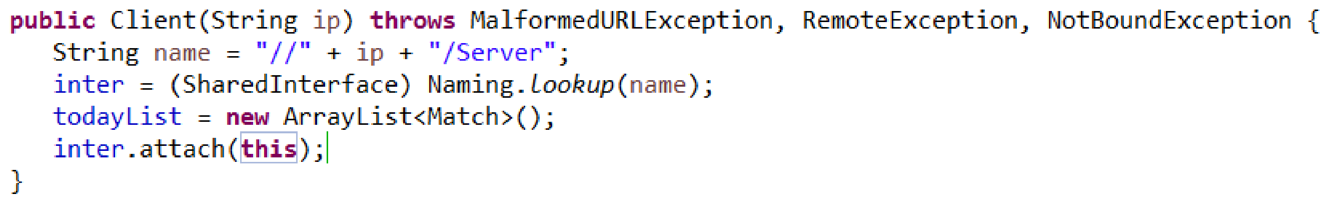
pkg
ControllerMain
- listener : ButtonListener
+ ControllerMain(model : Client, mainGui : MainGui)
+ setMouseAdapter() : MouseAdapter
- model
1
Client
- list : ArrayList<Match>
+ Client()
+ updateMatches() : void
+ getList() : ArrayList<Match>
+ getInter() : SharedInterface
1
- mainGui
- ticket
1
MainGui
- frame : JFrame
- tickets : JList<String>
- table : JTable
- historyTabel : JTable
- listenter : ActionListener
- adapter : MouseAdapter
- refreshAction : RefreshAction
- eventColumns : String[]
- ticketColumns : String[]
+ setListener(listenter : ActionListener) : void
+ setAdapter(adapter : MouseAdapter) : void
+ setRefreshAction(refreshAction : RefreshAction) : void
+ setTable(list : List<Match>) : void
+ setHisotyTable(data : Object[][]) : void
+ getType() : String
+ getDate1() : Date
+ getDate2() : Date
+ setScore(score : double) : void
+ setListModel(list : ArrayList<Ticket>) : void
+ setProfile(user : User) : void
+ setUser() : User
+ getRow() : Object[]
VirtualTicket
- frame : JFrame
- eventTable : JTable
- listener : ActionListener
- ticket : ArrayList<Object[]>
+ initialize() : void
+ setListener(listener : ActionListener) : void
+ addToTicket(obj : Object[]) : void
+ removeFromTicket() : void
+ resetTicket() : void
+ createTicket(user : User) : Ticket
+ setTable() : void
+ getMoney(max : double) : double
+ setVisibility(b : boolean) : void
+ addTicketSuccesfully() : void
+ emptyTicket() : void
1
- inter
<<interface>>
SharedInterface
+ attach(observer : IObserver) : void
+ insertUser(user : User) : boolean
+ availableUserName(userName : String) : boolean
+ verifyLogin(userName : String, password : String) : boolean
+ getUser(userName : String) : User
+ editUser(password : String, data : Data) : void
+ getBalance(userName : String) : void
+ setBalance(userName : String, d : double) : void
+ getMatch(matchId : int) : Match
+ getTodaysMatches() : ArrayList<Match>
+ getMatches(type : String) : ArrayList<Match>
+ getMatches(startDate : Date, endDate : Date) : ArrayList<Match>
+ addTicket(ticket : Ticket) : void
+ getTickets(user : User) : ArrayList<Ticket>
+ setTicketResult(id : int, value : boolean) : void
+ updateMatchesInTicket(ticket : Ticket) : void

Figure 17. MVC UML diagram

7.3.1 Model

A model captures the behavior of the application in terms of its problem domain, independent of the user interface. It directly manages the data, logic and rules of the system, it stores data that is retrieved according to commands from the controller and displayed in the view. The application logic is stored in

the “Client”, class which gathered the most important data about a user and has the SharedInterface from the server.
The following snippet of code shows the Client (model) constructor which gets all important methods from the server via the SharedInterface.



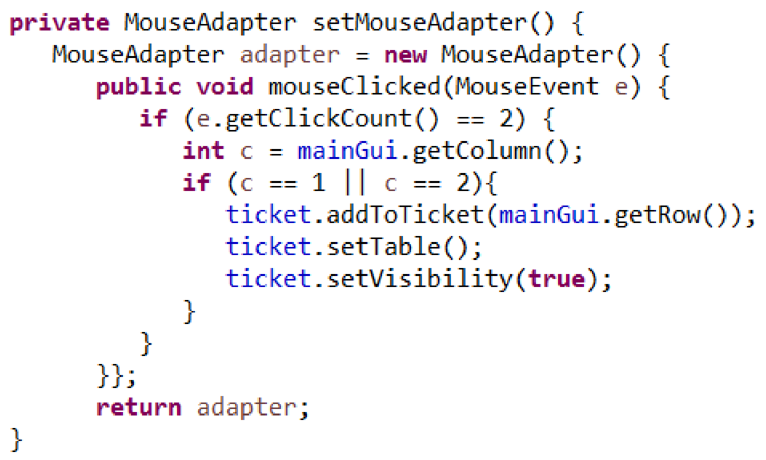
```
public Client(String ip) throws MalformedURLException, RemoteException, NotBoundException {
    String name = "//" + ip + "/Server";
    inter = (SharedInterface) Naming.lookup(name);
    todayList = new ArrayList<Match>();
    inter.attach(this);
}
```

7.3.2 View

The view consists of 2 classes (“MainGui” and “VirtualTicket”), which generate new output to the user based on changes in the model. Together with the model, it creates the core of the client application, which is controlled by the controller. The view element deals with the presentation of data to the user and for taking correct information given by the user. View examples are displayed in the next section.

7.3.3 Controller

The third part - the controller, accepts input from the view and converts it to commands for the model. It translates events into actions. The controller sends commands to the model to update the user or match states and it also sends commands to its associated view to change the view’s presentation of the model. The controller class contains one button action listener, event handlers for manipulation with the mouse and listeners for text fields. The button action listener consists of one big switch for all possible button action commands. For every case there is a corresponding action.



```
private MouseAdapter setMouseAdapter() {
    MouseAdapter adapter = new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
            if (e.getClickCount() == 2) {
                int c = mainGui.getColumn();
                if (c == 1 || c == 2){
                    ticket.addToTicket(mainGui.getRow());
                    ticket.setTable();
                    ticket.setVisibility(true);
                }
            }
        }};
        return adapter;
}
```

Figure 18. Mouse Adapter

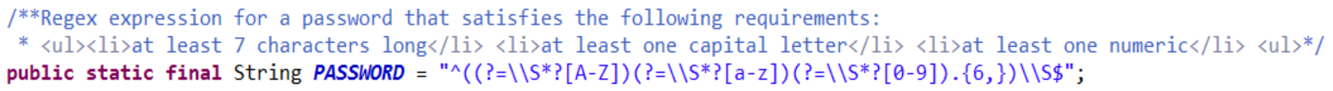
A crucial feature in this system is generating a user's virtual ticket which serves as a container for multiple events. This is done by double clicking on the table, which displays various events based on the search filters. This selection is handled with the controller, which takes data from the view and stores it in the model.

8. Input Validation

8.1 Regex

The input validation is possible with the java.util.regex package for pattern matching with regular expressions. Using the Pattern and Matcher classes from the aforementioned package, we are able to define a value domain for each piece information the user has to provide. By doing this, the chances of data discrepancies occurring when manipulating this data or later during data querying are reduced significantly. The same could be said for storing that same data in the database.

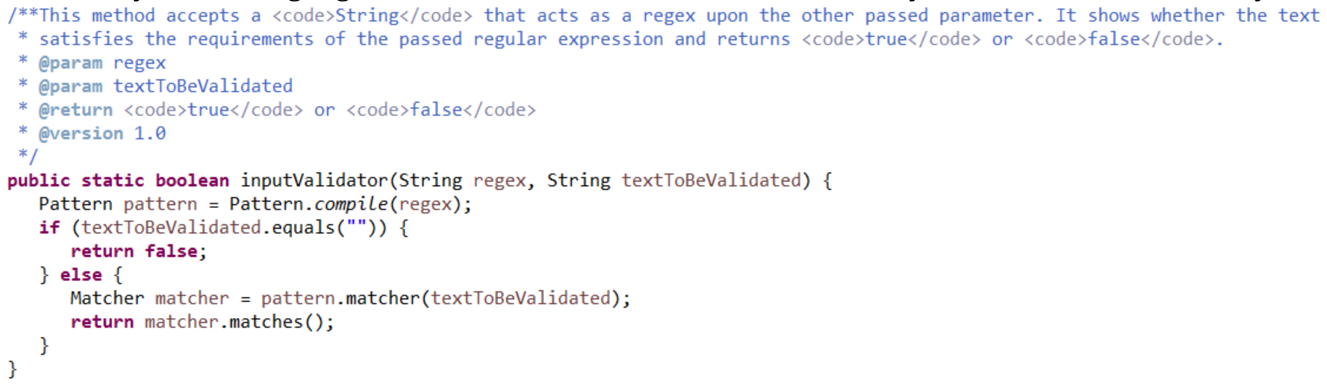
The domains are set by writing a regular expression, a special sequence of characters, which in this case is used to match a string entered by the User, and determine whether that string satisfies the regex requirements, as well as if it is a part of the specified domain. Below is an example of such a regex that ensures that the User has enter a valid password.



```
/**Regex expression for a password that satisfies the following requirements:
 * <ul><li>at least 7 characters long</li> <li>at least one capital letter</li> <li>at least one numeric</li> <ul>*/
public static final String PASSWORD = "^((?=\\S*?[A-Z])(?=\\S*?[a-z])(?=\\S*?[0-9]).{6,})\\S$";
```

Figure 19. Regex for a password

The RegexChecker class in the Utilities package contains not only the regexes that were created during the development of the system but also the method inputValidator(), that is created for reducing redundancy when using regex checks and also to increase the reusability of this whole functionality.

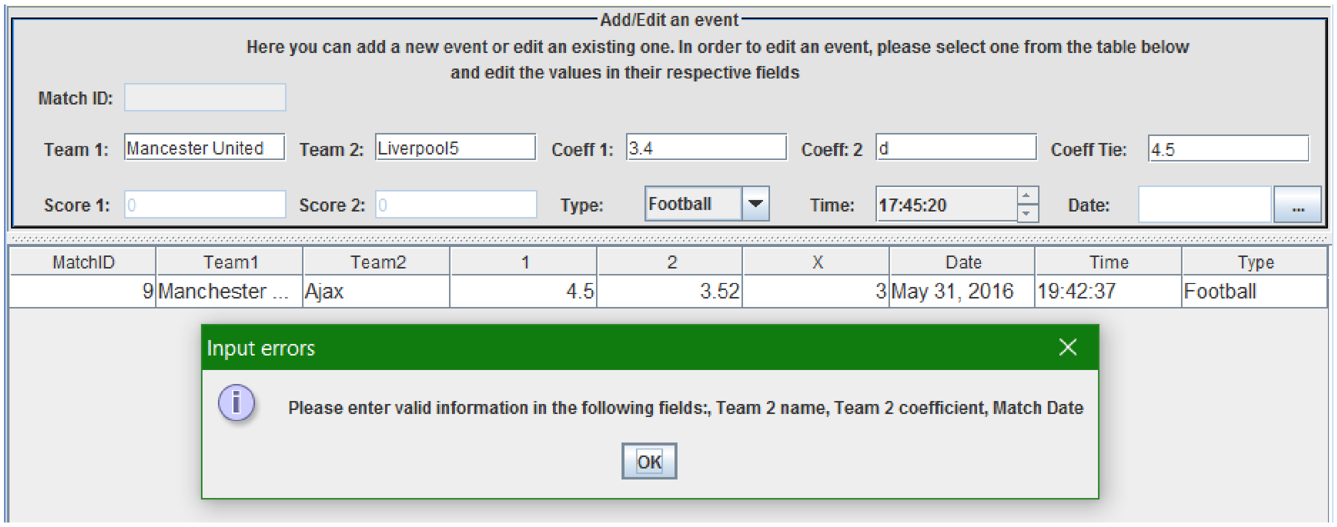


```
/**This method accepts a <code>String</code> that acts as a regex upon the other passed parameter. It shows whether the text
 * satisfies the requirements of the passed regular expression and returns <code>true</code> or <code>false</code>.
 * @param regex
 * @param textToBeValidated
 * @return <code>true</code> or <code>false</code>
 * @version 1.0
 */
public static boolean inputValidator(String regex, String textToBeValidated) {
   Pattern pattern = Pattern.compile(regex);
   if (textToBeValidated.equals("")) {
      return false;
   } else {
      Matcher matcher = pattern.matcher(textToBeValidated);
      return matcher.matches();
   }
}
```

Figure 20. inputValidator() method

8.2 Input Validator

The ValidatorFunctions class in the Utilities package is in itself a reusable system, implemented in several places in the project. It acts as an input validation assistant by showing the user all the text fields where information is not in the correct format or is not in the domain that the corresponding regex specifies.


Add/Edit an event

Here you can add a new event or edit an existing one. In order to edit an event, please select one from the table below and edit the values in their respective fields

Match ID:					
Team 1: Mancester United	Team 2: Liverpool5	Coeff 1: 3.4	Coeff: 2 d	Coeff Tie: 4.5	
Score 1: 0	Score 2: 0	Type: Football	Time: 17:45:20	Date:	

MatchID	Team1	Team2	1	2	X	Date	Time	Type
9	Manchester ...	Ajax	4.5	3.52	3	May 31, 2016	19:42:37	Football

Input errors

Please enter valid information in the following fields:, Team 2 name, Team 2 coefficient, Match Date

Figure 21. input validation

To achieve this functionality, the ValidatorFunction class works with regexes from the RegexChecker class described earlier. It is important to note that this whole verification function is carefully designed so, it could be reused at several places of the project, thus reducing the amount of code as well as facilitate the possible implementation of this subsystem in future updates or added functionalities. Below is a part of the code that makes the validation in the picture above possible.

```
validator.placeStringsAndExec(RegexChecker.TEAM_NAME, gui.getTextFieldTeam1().getText(), ValidatorFunctions.Fields.TEAM_1_NAME);
validator.placeStringsAndExec(RegexChecker.TEAM_NAME, gui.getTextFieldTeam2().getText(), ValidatorFunctions.Fields.TEAM_2_NAME);
validator.placeStringsAndExec(RegexChecker.COEFFICIENT, gui.getTextFieldCoeff1().getText(), ValidatorFunctions.Fields.TEAM_1_COEFFICIENT);
validator.placeStringsAndExec(RegexChecker.COEFFICIENT, gui.getTextFieldCoeff2().getText(), ValidatorFunctions.Fields.TEAM_2_COEFFICIENT);
validator.placeStringsAndExec(RegexChecker.COEFFICIENT, gui.getTextFieldCoeffTie().getText(), ValidatorFunctions.Fields.COEFFICIENT_TIE);
validator.placeStringsAndExec(gui.getDatePickerPanelEditField().getText(), ValidatorFunctions.Fields.MATCH_DATE);
```

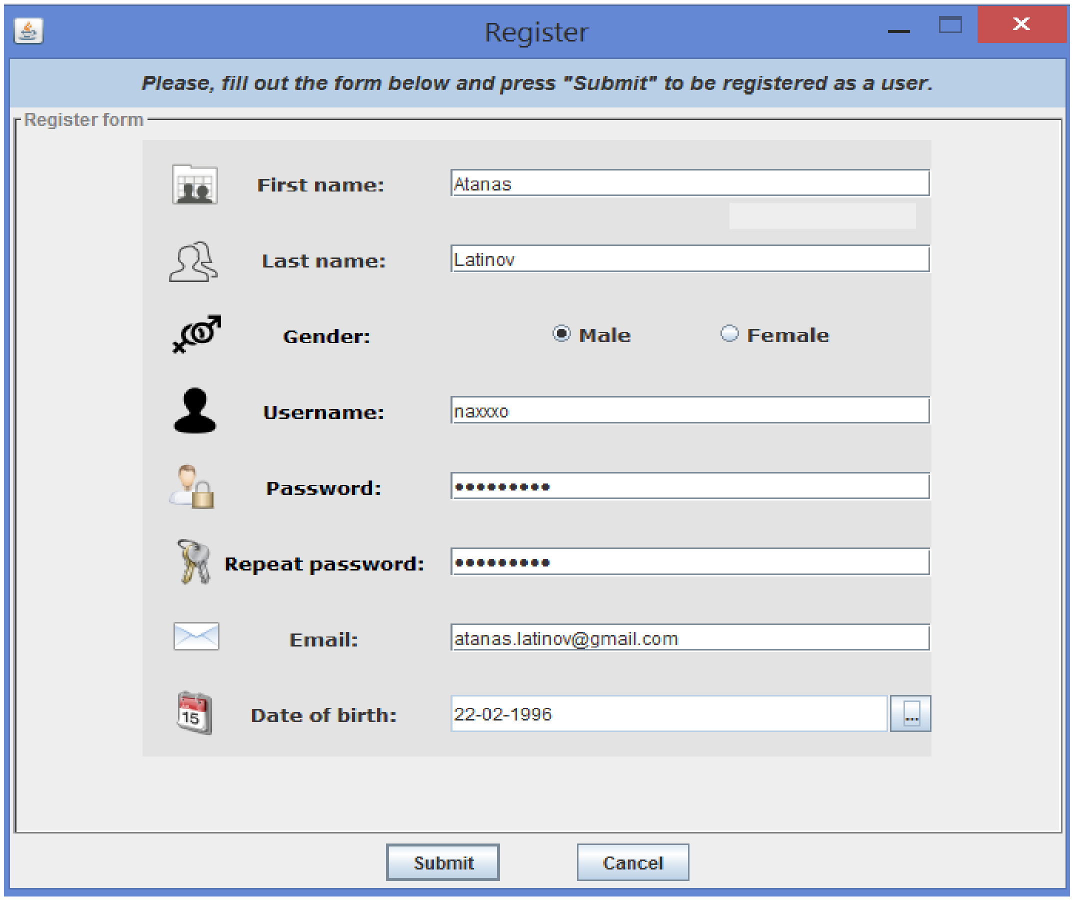
Figure 22. validator with regex

The method placeStringsandExec() creates the flexibility of the validation implementation as it gives the developer the freedom to place a regex of their choice as well as text field that contains the data they want to test with that regex.

9. Design – User's interface

9.1 Registration form

The image below presents the entry point for users who are not registered in the system. Here they enter their personal information, as well as specify their username and password, which act as unique identifiers of each account in the system. To make sure that the user is entering data in the valid format, input validation is going through each information field on pressing the Submit button.



Please, fill out the form below and press "Submit" to be registered as a user.

Register form

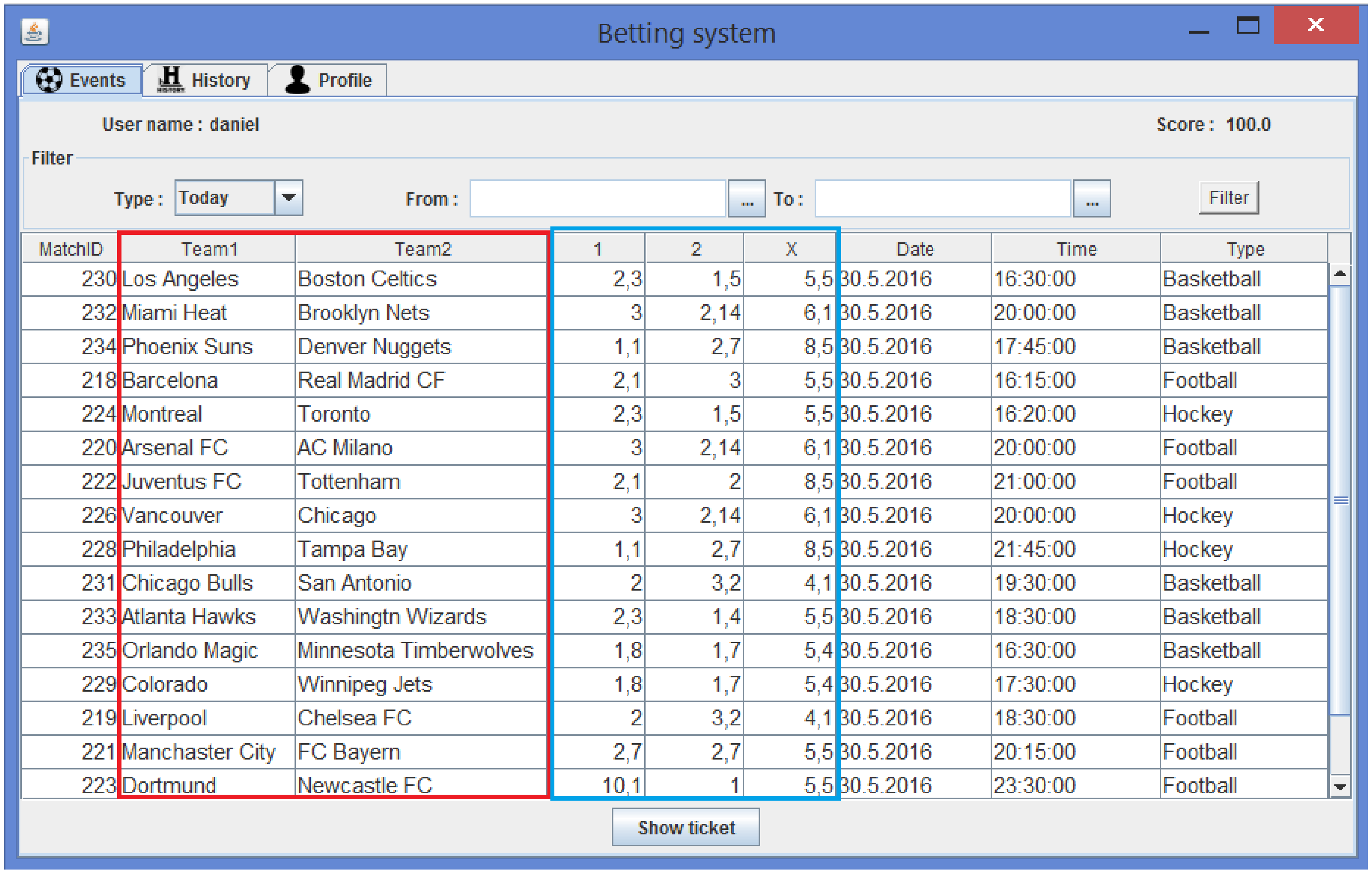
First name:	Atanas
Last name:	Latinov
Gender:	[✓ Male] Female
Username:	naxxxo
Password:	•••••••••
Repeat password:	•••••••••
Email:	atanas.latinov@gmail.com
Date of birth:	22-02-1996

Figure 23. Registration form

9.2 Betting system GUI

After the user logs in/registers, the main window of the program appears. What is interesting about it is that it contains 3 tabbed panes.
The first pane named Events presents the user with the ability to filter a multitude of events by type, start and end date. After he/she specifies their filtering criteria, they can select matches of their choice and then browse them in a virtual betting ticket which they can see by pressing the button show ticket.

The core of the adding mechanic is that the table is divided into sectors and bets can be placed depending on where the user clicks. The team names system makes it possible to add a match, including the bet on that match, to the virtual ticket, by double clicking on a team's name. The other way to place a bet is to double-click on the team's coefficient, which leads to the same result – placing the match as well as its respective bet in the virtual ticket.



User name : daniel

Score : 100.0

Type : Today

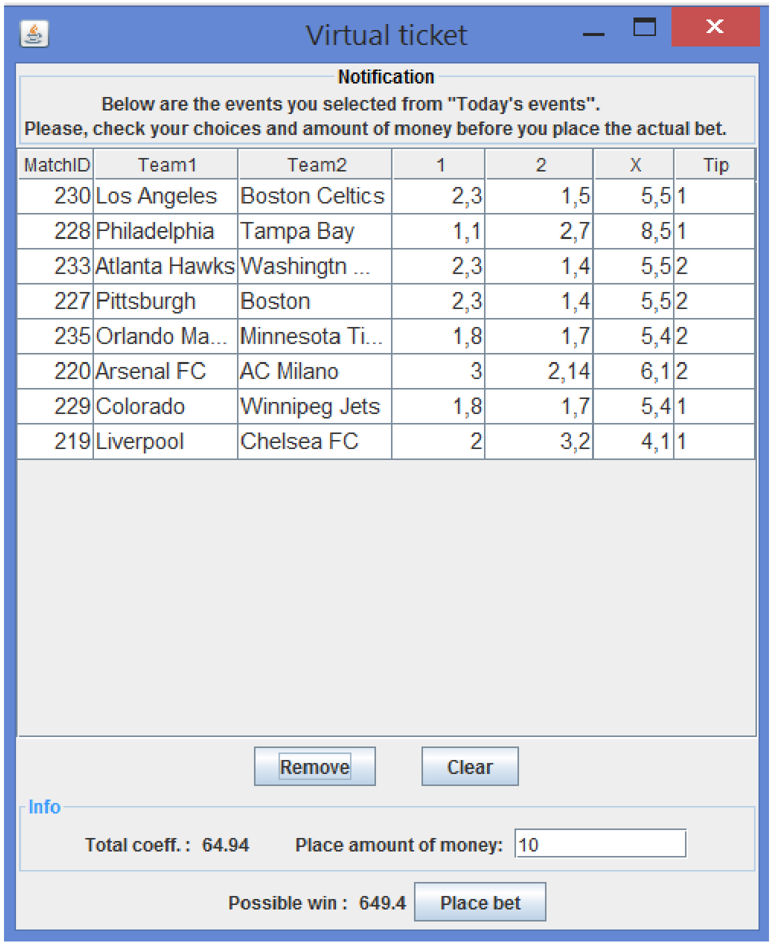
From :

To :

MatchID	Team1	Team2	1	2	X	Date	Time	Type
230	Los Angeles	Boston Celtics	2,3	1,5	5,5	30.5.2016	16:30:00	Basketball
232	Miami Heat	Brooklyn Nets	3	2,14	6,1	30.5.2016	20:00:00	Basketball
234	Phoenix Suns	Denver Nuggets	1,1	2,7	8,5	30.5.2016	17:45:00	Basketball
218	Barcelona	Real Madrid CF	2,1	3	5,5	30.5.2016	16:15:00	Football
224	Montreal	Toronto	2,3	1,5	5,5	30.5.2016	16:20:00	Hockey
220	Arsenal FC	AC Milano	3	2,14	6,1	30.5.2016	20:00:00	Football
222	Juventus FC	Tottenham	2,1	2	8,5	30.5.2016	21:00:00	Football
226	Vancouver	Chicago	3	2,14	6,1	30.5.2016	20:00:00	Hockey
228	Philadelphia	Tampa Bay	1,1	2,7	8,5	30.5.2016	21:45:00	Hockey
231	Chicago Bulls	San Antonio	2	3,2	4,1	30.5.2016	19:30:00	Basketball
233	Atlanta Hawks	Washingtn Wizards	2,3	1,4	5,5	30.5.2016	18:30:00	Basketball
235	Orlando Magic	Minnesota Timberwolves	1,8	1,7	5,4	30.5.2016	16:30:00	Basketball
229	Colorado	Winnipeg Jets	1,8	1,7	5,4	30.5.2016	17:30:00	Hockey
219	Liverpool	Chelsea FC	2	3,2	4,1	30.5.2016	18:30:00	Football
221	Manchaster City	FC Bayern	2,7	2,7	5,5	30.5.2016	20:15:00	Football
223	Dortmund	Newcastle FC	10,1	1	5,5	30.5.2016	23:30:00	Football

Figure 24. Main GUI

After the users are done with their picks, they can see a summary of their betting activity by clicking on the show ticket button.


Virtual ticket

Notification

Below are the events you selected from "Today's events".
Please, check your choices and amount of money before you place the actual bet.

MatchID	Team1	Team2	1	2	X	Tip
230	Los Angeles	Boston Celtics	2,3	1,5	5,5	1
228	Philadelphia	Tampa Bay	1,1	2,7	8,5	1
233	Atlanta Hawks	Washingtn ...	2,3	1,4	5,5	2
227	Pittsburgh	Boston	2,3	1,4	5,5	2
235	Orlando Ma...	Minnesota Ti...	1,8	1,7	5,4	2
220	Arsenal FC	AC Milano	3	2,14	6,1	2
229	Colorado	Winnipeg Jets	1,8	1,7	5,4	1
219	Liverpool	Chelsea FC	2	3,2	4,1	1

Total coeff. : 64.94 Place amount of money: 10

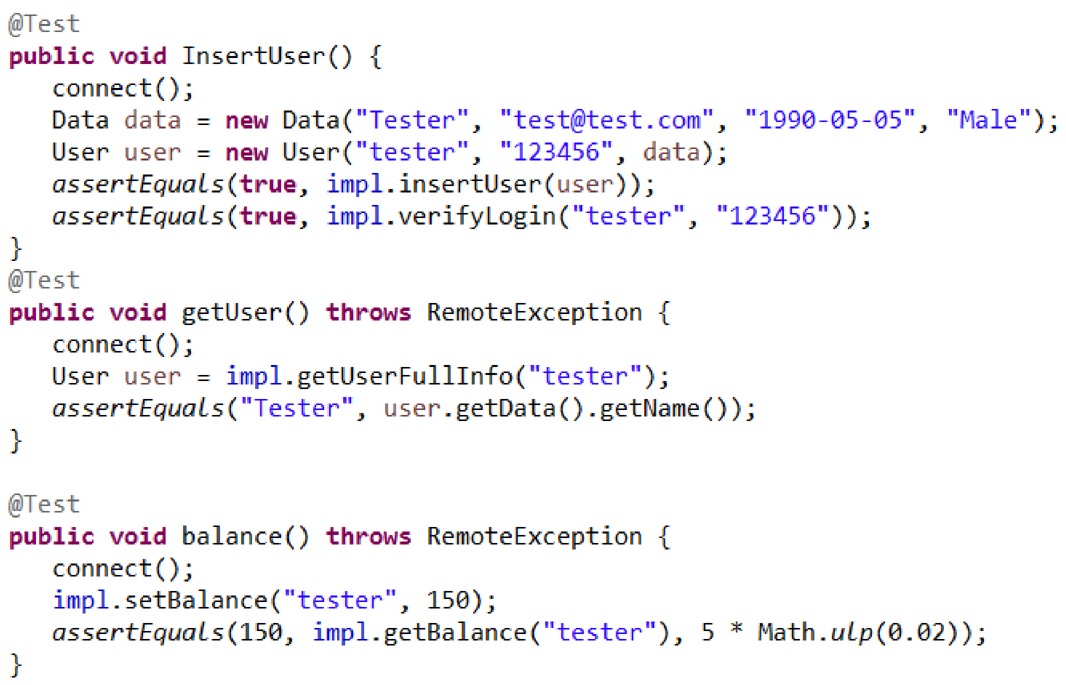
Possible win : 649.4

Figure 25. Virtual ticket

The virtual ticket stores all of the users picks from their current betting session. Not only does it provide them with information about the selected matches but also it shows them the maximum possible amount of points they can win. They also have the ability to remove events they selected or clear the whole virtual ticket. By clicking on the "Place bet" button, they have made a new pending entry in their betting history which will be updated in the future depending on the selected matches' results. The user score will also be affected, depending on the outcome of the matches.

10. Testing

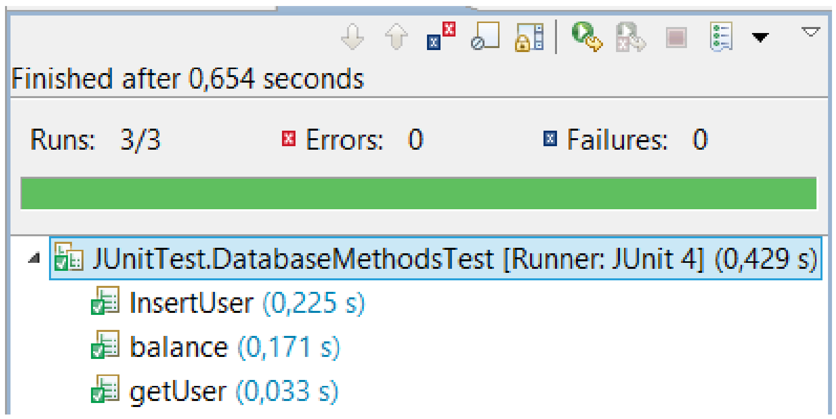
A variety of methods have been used to ensure that every functional and non-functional requirement in the system has been tested. Junit tests have been used to test certain classes in the client and server models, since some of the classes in the system were too difficult to write unit tests for, such as the Model-view-controller design pattern. End to end testing was conducted using debugging methods as well as cross-checking the database with the client to evaluate if the data was being handled correctly. The following JUnit test proved that registration of a new user, reading from database and updating is correct.



```
@Test
public void InsertUser() {
    connect();
    Data data = new Data("Tester", "test@test.com", "1990-05-05", "Male");
    User user = new User("tester", "123456", data);
    assertEquals(true, impl.insertUser(user));
    assertEquals(true, impl.verifyLogin("tester", "123456"));
}
@Test
public void getUser() throws RemoteException {
    connect();
    User user = impl.getUserFullInfo("tester");
    assertEquals("Tester", user.getData().getName());
}

@Test
public void balance() throws RemoteException {
    connect();
    impl.setBalance("tester", 150);
    assertEquals(150, impl.getBalance("tester"), 5 * Math.ulp(0.02));
}
```

Figure 26. JUnit test



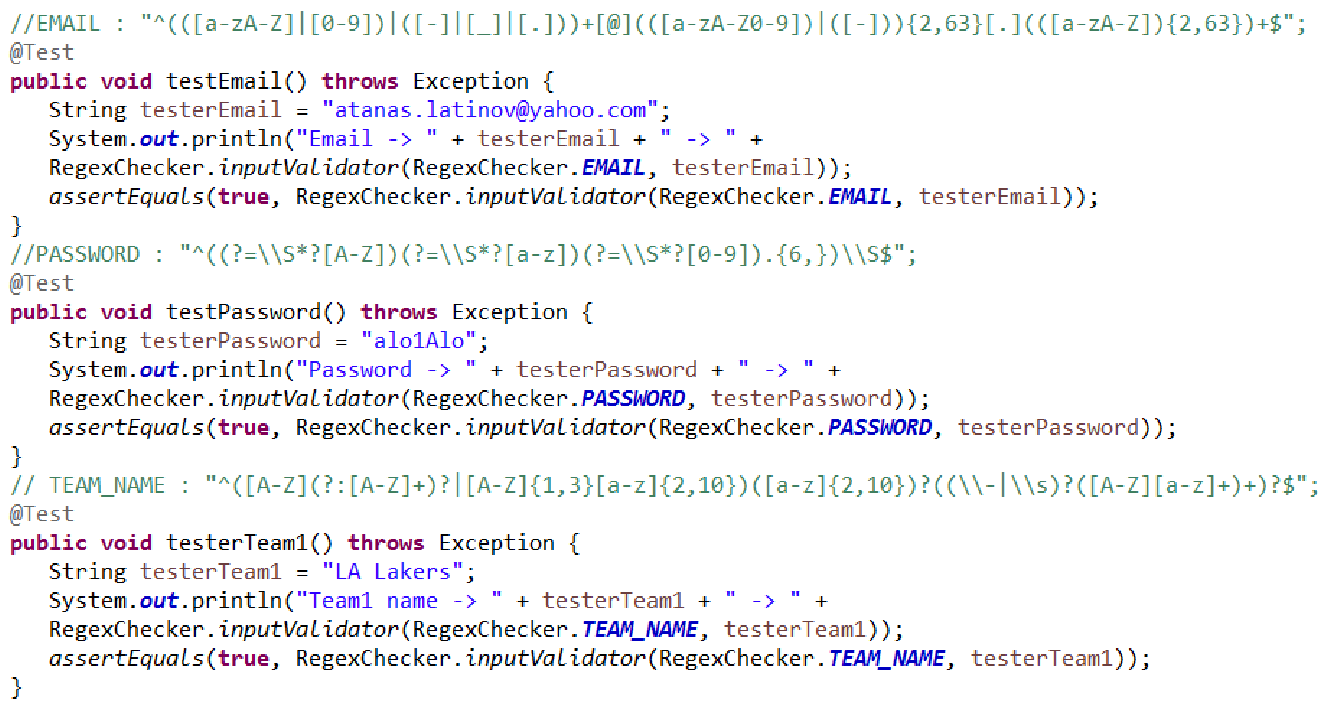
```
Finished after 0,654 seconds

Runs:   3/3          Errors:   0          Failures:   0

JUnitTest.DatabaseMethodsTest [Runner: JUnit 4] (0,429 s)
    InsertUser (0,225 s)
    balance (0,171 s)
    getUser (0,033 s)
```

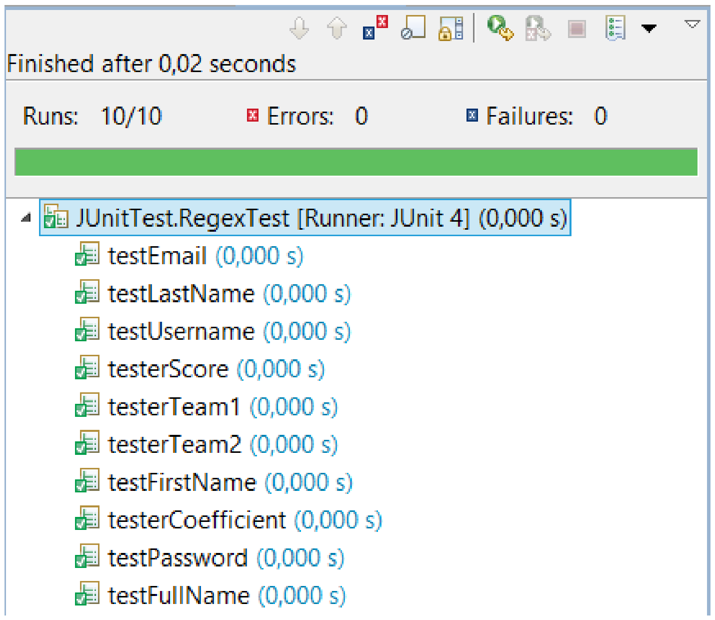
Figure 27. JUnit result

Validation of users' inputs is one of the crucial functions for avoiding errors in the system. The Validation class, which is responsible for almost all inputs in the program is based on regular expression (Regex). The following series of tests confirmed that the regex and validation classes are fully functional.



```
//EMAIL : "^(([a-zA-Z]|[0-9])|([-]|[_]|[.]))+[@](([a-zA-Z0-9])|([-])){2,63}[.](([a-zA-Z]){2,63})+$";
@Test
public void testEmail() throws Exception {
    String testerEmail = "atanas.latinov@yahoo.com";
    System.out.println("Email -> " + testerEmail + " -> " +
    RegexChecker.inputValidator(RegexChecker.EMAIL, testerEmail));
    assertEquals(true, RegexChecker.inputValidator(RegexChecker.EMAIL, testerEmail));
}
//PASSWORD : "^((?=\\S*?[A-Z])(?=\\S*?[a-z])(?=\\S*?[0-9]).{6,})\\S$";
@Test
public void testPassword() throws Exception {
    String testerPassword = "alo1Alo";
    System.out.println("Password -> " + testerPassword + " -> " +
    RegexChecker.inputValidator(RegexChecker.PASSWORD, testerPassword));
    assertEquals(true, RegexChecker.inputValidator(RegexChecker.PASSWORD, testerPassword));
}
// TEAM_NAME : "^([A-Z](?:[A-Z]+)?|[A-Z]{1,3}[a-z]{2,10})([a-z]{2,10})?((\\-|\\s)?([A-Z][a-z]+)+)?$";
@Test
public void testerTeam1() throws Exception {
    String testerTeam1 = "LA Lakers";
    System.out.println("Team1 name -> " + testerTeam1 + " -> " +
    RegexChecker.inputValidator(RegexChecker.TEAM_NAME, testerTeam1));
    assertEquals(true, RegexChecker.inputValidator(RegexChecker.TEAM_NAME, testerTeam1));
}
```

Figure 28. JUnit test - Regex



```
Finished after 0,02 seconds

Runs: 10/10        Errors: 0        Failures: 0

JUnitTest.RegexTest [Runner: JUnit 4] (0,000 s)
    testEmail (0,000 s)
    testLastName (0,000 s)
    testUsername (0,000 s)
    testerScore (0,000 s)
    testerTeam1 (0,000 s)
    testerTeam2 (0,000 s)
    testFirstName (0,000 s)
    testerCoefficient (0,000 s)
    testPassword (0,000 s)
    testFullName (0,000 s)
```

Figure 29. JUnit - Regex result

11. Result

All functional and non-functional requirements were met by the system, and all conducted tests were passed. A new user is able to create a new personal account, which is stored in the system's database. After the user is logged in, he/she can immediately place bets on selected matches. The Virtual ticket GUI helps keeping all selected matches organized and manage all selected tickets, total coefficients and amounts of score points. For the client-server connection we used RMI (Java Remote Method Invocation). With these design considerations, the client and server ends were able to connect over a network and call methods back and forth with as little complexity as possible. Manipulation with data is extremely simplified since everything is stored in the relational database and java is connected to the database with JDBC. Client end architecture is based on the MVC (model-view-controller) design pattern, which keeps everything structured and easy to orientate around. The model and the view are completely separated and connected via the controller. The client-specific data is held in-memory in the client model, and is synchronized with actual data in the database because of the Observer pattern.

12. Discussion

The thoroughly inspected design of the relational database provides great flexibility for the developers to include further functionalities in future updates. The various querying methods in the code, are designed so data extraction from the database is extremely simplified and intuitive. This consideration is beneficial for further upgrades of the current functionalities. Since data manipulation is made so effortless and allows for the addition of further user related actions, it is safe to say that the system is designed with a main focus on maintainability and reusability of the code. This is of huge importance and it should remain so. The reason for this focus is that, in order for the life-cycle of such a software product to be increased or at least maintained, the actual code should be clearly written and well-structured so new developers can quickly familiarize themselves with the various sections of the code and be easily introduced to the basis of the system. The bus factor is kept to a minimum which is imperative for the success of the project as it gives all of the participating developers the chance to make quality contributions to the existing software since they are familiar with all the parts of the program.

Apart from the database design, special attention is given to the code structure and design. Due to the implementation of several design patterns such as MVC, developers can quickly orient themselves around the code and debug much faster.

The tests that consisted of interacting with the different user interfaces, have proved that the user experience is intuitive and unambiguous. During those tests, the different use cases proved the stability of the system and showed how it reacts based on diverse input. The different user-action flows also narrow down the possible inefficiencies and subsequent activities do not seem to lead the system to unexpected crashes. Here we would like to acknowledge the fact that impeccable systems do not exist, so it is left to the users to find and report the bugs and errors they come across, no matter how impactful or small they are, so we can address the issues as quickly as possible.

Regarding further improvements or additions, the user interface could be made much more visually engaging and given the popularity of mobile devices and their impact on the world, a mobile counterpart of the system will definitely be a bold step towards upscaling the project and making it available to millions of users.

13. Conclusion

The aim of the project was to design and create a system which works over a network (client-server connection) and stores all data digitally in a relational database. The betting system’s architecture is also based on several design patterns, which ensures a very clear and readable structure. The system enables its users to create their own personal accounts, which monitor their progress and results of every match. The registration form is protected against wrong input with an embedded input validation system. After the registration/log in process, a user can immediately see upcoming events in the event table. The table can be filtered based on a type of event and a time period, which means that the user can easily orientate around a wide collection of events. When the user finds a match he/she wants to bet on, he/she can do that simply by double-clicking on the team or coefficient. When the user is done with the selection of matches, he can select a score amount he would like to bet. Everything is displayed in the virtual ticket, which helps the user keep track on all selected events, the total coefficient and amount of score he/she wants to spend. After the confirmation of the ticket, everything is stored in the system’s database and later evaluated, according to the match results. When bets are correct, the amount of score placed with the bet, is multiplied by the total coefficient and the result is added to the user’s actual score. Users can keep track on all of their bets in the betting history as well as see detailed information about every match and its result. Users are also able to edit their personal or contact information in their profile. The end result is an extremely stable application that satisfies all functional and non-functional requirements, offers a great deal of functionality and has a very user-friendly interface that complies with the latest software standards. In conclusion, the goals of the project are met and all the demands are fulfilled.

References

Anon n.d. *Java RMI Release Notes*. Available at: <http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/relnotes.html> [Accessed 21 Apr. 2016].

Wollrath, A., Riggs, R. and Waldo, J., 1996. A Distributed Object Model for the Java System. *Coots*.

Ambler, S., 2002. *Agile Modeling: Effective Practices for EXtreme Programming and the Unified Process*. [online] J. Wiley. Available at: <http://www.ambysoft.com/books/agileModeling.html> [Accessed 27 Apr. 2016].

(Ambler, 2002)

Ambler, S. and Constantine, L., 2002. *The Unified Process Transition and Production Phases*. [online] CMP Books. Available at: <http://www.ambysoft.com/books/transitionProductionPhase.html> [Accessed 27 Apr. 2016].

Ambler, S., 2002. *Agile Modeling: Effective Practices for EXtreme Programming and the Unified Process*. [online] J. Wiley. Available at: <http://www.ambysoft.com/books/agileModeling.html> [Accessed 27 Apr. 2016].

(Ambler, 2002)

(Ambler and Constantine, 2002)

(Generic Methods (The Java^{TM} Tutorials > Bonus > Generics), 2016)

Powell, G., 2006. *Chapter 8: Building Fast-Performing Database Models*. *Beginning Database Design ISBN 978-0-7645-7490-0*. Available at: <http://searchsecurity.techtarget.com/generic/0,295582,sid87_gci1184450,00.html> [Accessed 27 Apr. 2016].

Powell, G., 2006. *Chapter 8: Building Fast-Performing Database Models*. *Beginning Database Design ISBN 978-0-7645-7490-0*. Available at: <http://searchsecurity.techtarget.com/generic/0,295582,sid87_gci1184450,00.html> [Accessed 27 Apr. 2016].