

# **fundamniti!**

## **Group 14**

Per Almquist  
Peter Andersson  
Marcus Bergenlid  
Victor Mangs  
Ali Mosavi

# Contents

1. Introduction.....	4
1.1 Summary of the document .....	4
2. System Overview .....	5
2.1 General description .....	5
2.2 Overall Architecture Description.....	5
2.3 Detailed Architecture.....	6
2.3.1 Server Architecture .....	6
2.3.1.1 Server Architecture Modules .....	6
2.3.1.2 Server architecture control and data flow .....	7
2.3.2 Client architecture .....	9
2.3.2.1 Client architecture modules .....	9
2.3.2.2 Client architecture control and data flow .....	10
3. Design Considerations .....	11
3.1 Assumptions and Dependencies .....	11
3.1.1 Related software .....	11
3.1.2 Related hardware .....	11
3.1.3 End-user characteristics .....	12
3.1.4 Possible and/or probable changes in functionality .....	12
3.2 General Constraints.....	12
4. Graphical User Interface .....	13
4.1 Overview of the User Interface .....	13
4.2 Graphical user interface forms.....	14
4.2.1 Free Sketch .....	14
4.2.2 Global gallery .....	14
4.2.3 Home.....	15
4.2.4 Login.....	15
4.2.5 Profile .....	16
4.2.6 Playing arena .....	17
4.2.7 Search .....	18
4.2.8 Sign up.....	19
4.2.9 Vote .....	20
5. Design Details.....	21
5.1 Class Responsibility Collaborator (CRC) cards.....	21
5.1.1 View.....	21
5.1.2 Controller.....	22
5.1.3 Model.....	24
5.2 Class diagram .....	26
5.2.1 View.....	26
5.2.2 Model.....	26
5.2.3 Controller.....	27
5.2.4 Playing Arena .....	28
5.3 State charts .....	29
5.4 Interaction diagrams .....	30
5.4.1 Register .....	30
5.4.2 Login.....	31
5.4.3 Challenge .....	32
5.4.4 Battle.....	33
5.4.5 Vote .....	34
5.4.6 Add guestbook message .....	35
5.5 Detailed Design.....	36
5.6 Package diagram.....	167

6. Test cases.....	168
6.1. Register .....	168
6.2. Login.....	169
6.3. Artistic points .....	170
6.4. Guestbook.....	171
6.5. Personal gallery.....	172
6.6. Statistics .....	173
6.7. Playing arena .....	174
6.8. Playing arena idle.....	175
6.9. Signup .....	176
6.10. Show challengeable users .....	177
6.11. Chat room .....	178
6.12. Challenging users .....	179
6.13. Drawing board colors .....	180
6.14. Painting tools - Pencil .....	181
6.15. Painting tools - bucket .....	182
6.16. Free sketch mode .....	183
6.17. Competition .....	184
6.18. Voting .....	185
6.19. Golden Vote.....	186
6.20. AP transfer .....	187
6.21. Challenge options – Time limit .....	188
6.22. Challenge options – Bet.....	189
6.23. Challenge request.....	190
6.24. AP reservation.....	191
6.25. Battle topic .....	192
6.26. Vote time limit.....	193
6.27. Voting page .....	194
6.28. Vote weight .....	195
6.29. Vote statistics .....	196
6.30. Voter gets AP, can only vote once and not in own competition .....	197
6.31. Top ten user list.....	198
6.32. Global gallery .....	199
6.33. Search users .....	200

# 1. Introduction

This document is intended for developers of the fundamnit! system. The purpose is to give the developer an insight into how the system will look and feel and how the various components of the system are supposed to interact with each other.

After reading this document the developer should be able to implement the system in a way that satisfies the initial requirements that were put on the system.

This document describes a design of the fundamnit! system version 1.0.

The reader of this document should have read or at least have the possibility to concurrently read the requirements document that this design document is based on, since there are a lot of references to that document.

For important terms, acronyms and abbreviations, see the requirements document section 3 (Glossary).

## 1.1 Summary of the document

The document starts out with a walkthrough of the overall architecture of the system in section 2. After that, some general aspects of the design that needs special attention is discussed in section 3.

In section 4 we tackle some guidelines about the graphical user interface that must be taken into account.

In section 5 we go into the core of the system and describe the detailed design of all classes and packages in the system. Accompanying these are a lot of detailed diagrams that should give the developer a firm ground to stand on when the implementation is being planned. Also included in section 5 are a lot of interaction diagrams that further describe some of the thoughts behind the design.

In section 6, some test cases are described that can be used after or during implementation to make sure that the system functions as intended.

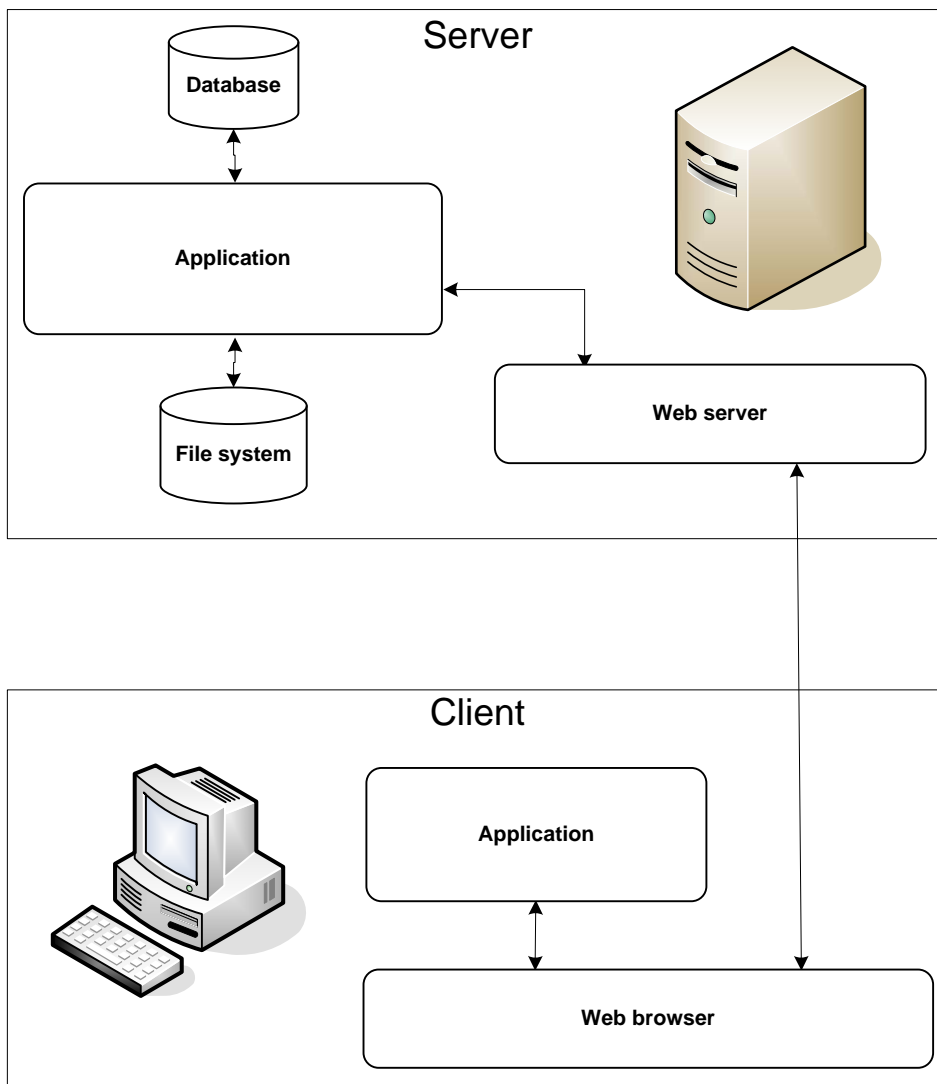
## 2. System Overview

### 2.1 General description

Fundamnit! is a web based community that focuses on user competing in the art of painting. The system is intended to be entertaining but may also serve as a natural meeting place for users across the globe.

The system will be a client and server application, the server will run a Tomcat web server so the interface between the client and server will follow the HTTP protocol. The client side will consist of a web browser (Mozilla Firefox should be used for the best result) and Adobe Flash Player.

### 2.2 Overall Architecture Description



The general architecture applied is a two tier (semi) fat client web service system, meaning that the server basically does nothing more than receive requests, processes them and sends back the

requested information. Semi-fat means that there's an event driven application on the client side that does some of the html generation and injection. The actual pages being viewed aren't fully complete (at least in some cases), the missing information is fetched and processed real time when the page is loading and/or as a response to user interaction.

On the server side there exists a web server which handles the low level http protocol requests and reroutes them to the web service application which decides how to respond. The application may also request information from the database or from a separate file.

On the client side a web browser (supporting XHTML, Flash plugin, ECMAScripts) handles the requests to the server and presents the results to the end user. In aid of presenting the information and interacting with the user, the browser uses various applications.

## 2.3 Detailed Architecture

In this section a more detailed version of the architecture is explained.

### 2.3.1 Server Architecture

The server as a whole is made out of several major components as depicted in the figure in section 2.3.1.2. Each of the individual components will be described in the next section followed by a control and data flow description in section 2.3.1.2.

#### 2.3.1.1 Server Architecture Modules

##### Tomcat

Apache Tomcat<sup>1</sup> is a web server derived from Apache Hypertext Transfer Protocol (HTTP) Server<sup>2</sup> which adds support for Hypertext Transfer Protocol over Secure Socket Layer (HTTPS) protocol as well as being the standard implementation for JEE Web Application Technologies.

##### Java Enterprise Edition (JEE) Web Application Technologies

The JEE Web Application Technologies<sup>3</sup> is a collection of technologies for Java based web applications. Two of these are Java Servlets and Java Server Pages (JSP) which we will be using extensively. It is basically a layer or interface linking the web server with the custom written Java code.

When an incoming http request (which is mapped to some servlet) is received by Tomcat it is forwarded to the JEE Web Application technologies which will handle the details concerning the http request and then call the correct custom Servlet. The JEE Web services are run inside/on top of the Java Virtual Machine (JVM).

---

<sup>1</sup> <http://tomcat.apache.org/>

<sup>2</sup> <http://httpd.apache.org/>

<sup>3</sup> <http://java.sun.com/javaee/technologies/webapps/>

## **Service Application**

The service application is a collection of custom written Servlets and JSP pages that will receive different requests made by the user in the form of HTTP requests, process them and if required, return the appropriate information. It is here that all of the server side logic of our software system resides. It will use the database (through Hibernate) to store all of the data concerning the users and the system, and the file system to store the user drawings.

## **Hibernate**

Hibernate<sup>4</sup> is a sort of middleware which will simplify the integration of a Structured Query Language (SQL) database with the rest of the application by acting as a link between the programming language and the database, doing on the fly (two way) translation between Java objects and SQL. In simplified terms this means that it will automatically transform Java objects to SQL queries and database table rows. It will also do the opposite, that is, transform the result of a query back to Java objects. This will simplify the development of a data intense application such as this as the code need not to be cluttered with SQL code and will also greatly simplify the evolution of the data model as the code more or less needs not to be aware of the database schema.

## **JVM**

The JVM will run all the java related code. It needs to comply to the Java 1.5 standard as Generics will be used.

## **The database**

The database will be a standard PostgreSQL<sup>5</sup> setup which need not be on the same physical machine as the Apache Tomcat server.

## **The file system**

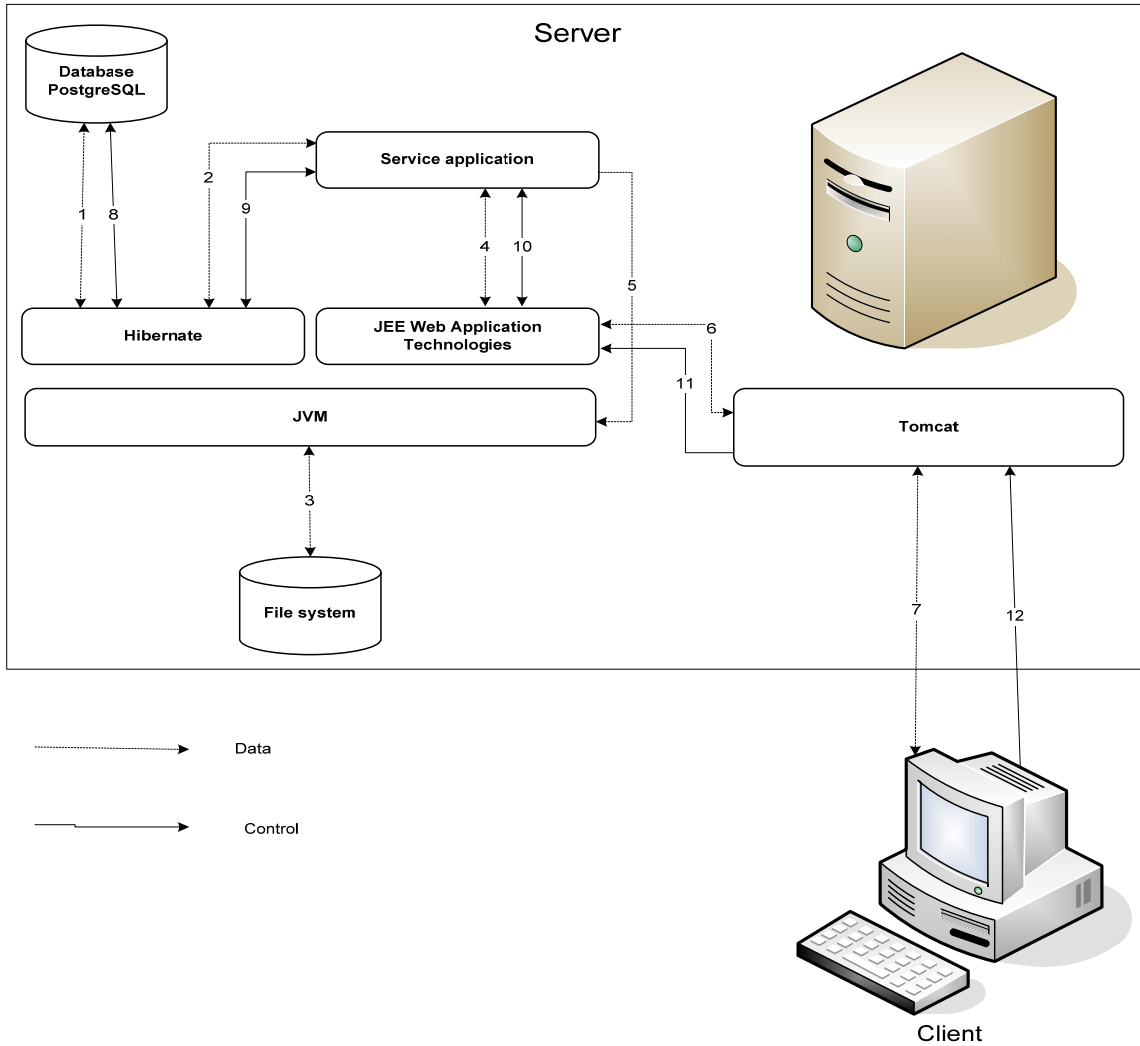
The underlying file system will be used to store the user created pictures. As the server side application is entirely written in Java, it is independent of the platform. So the only requirement on the file system is that it supports long file names (up to 256 characters) and can be a network mapped drive.

### **2.3.1.2 Server architecture control and data flow**

---

<sup>4</sup> <http://www.hibernate.org/>

<sup>5</sup> <http://www.postgresql.org/>



This section will provide a general and simplified overview of the control and data flow of the architecture model. The text will often be followed by a number inside a parenthesis. This number is a reference to the numbered arrows in the figure above. The different steps in this process are described below.

1. The client will make HTTP/HTTPS requests to the server (12). In some cases (such as login) this request will be made synchronously, meaning that the control will be given over to the server and nothing more will occur until a response (7) has been received. But in most case it be an asynchronous request and the client can go on to do whatever it wishes.
2. Tomcat will handle this request and forward (11) it to JEE Web Application layer along with any additional data (6) in the HTTP request.
3. The JEE Application layer will in turn forward (10) this to the correct custom written Servlet or JSP along with the parsed extra data (4).
4. The custom servlet will process the request and gather any additional data required through Hibernate (9) which will return the data (2).



5. Hibernate will translate the request to SQL and send (8) a query to the database. It will receive the response (1) and transform it back to Java objects which will be returned (2) to the custom Servlet.
6. Having the required data, a correct response can now be constructed by the custom Servlet. It might require a specific file which will be fetched (5 and 3).
7. The response is now constructed by the Servlet and returned (4) to JEE Web Application layer which returns it to Tomcat (6), which in turn, forwards (7) it to the client.
8. The request is now complete.

## **2.3.2 Client architecture**

The client is also made out of several major components. Again, first a description of the components then a description of the control and data flow is described.

### **2.3.2.1 Client architecture modules**

#### **Web browser**

The web browser serves as the HTTP client that makes and handles HTTP requests and responses to and from the server. It views the html and host the JavaScript interpreter, and plugins such as Flash Player.

#### **Flash Player**

The Flash Player is an ActionScript engine that executes ActionScript source code.

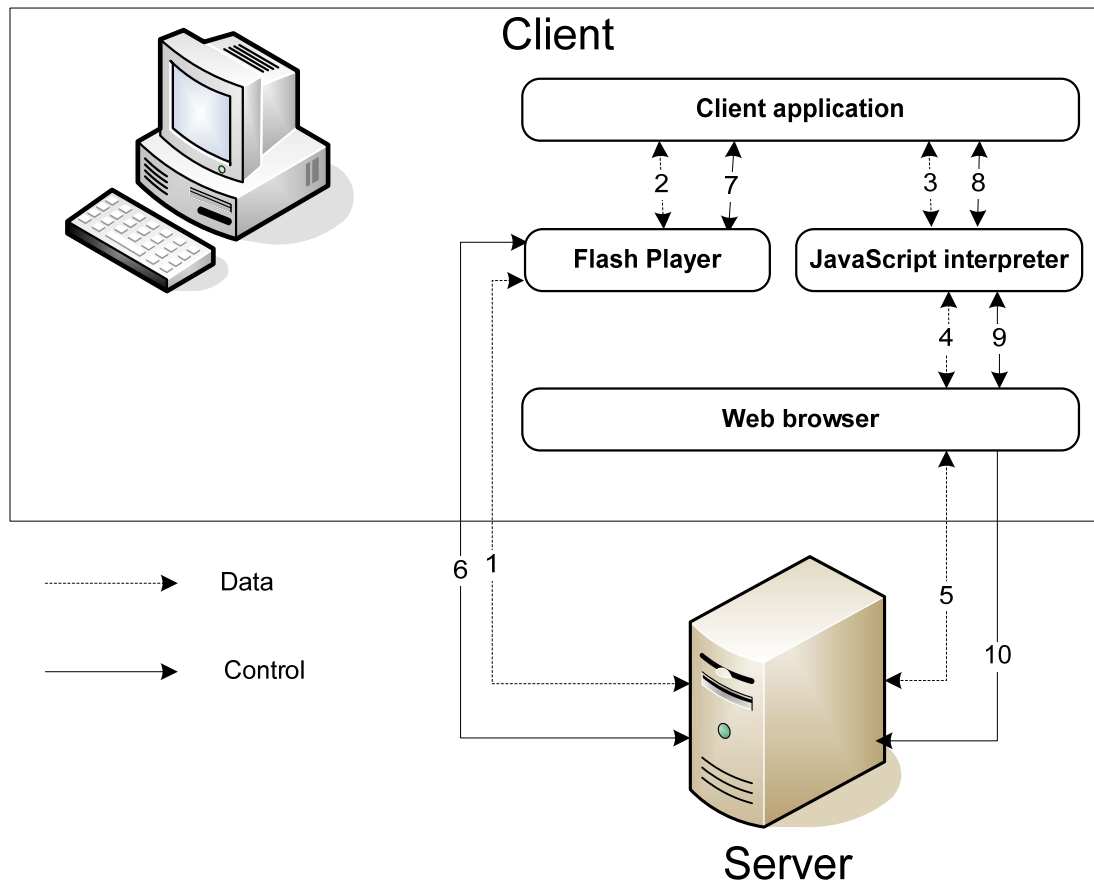
#### **JavaScript interpreter**

The JavaScript interpreter will run the actual client application, which handles different events and performs the corresponding action.

#### **Client application**

The client application is a collection of custom ECMAScripts written mainly in JavaScripts with the exception of the one painting component that is written in ActionScript. It is here that all the client side logic of the system resides. Takes care of the user interaction with the system and generates HTML for different actions.

### 2.3.2.2 Client architecture control and data flow



This section corresponds to section 2.3.1.2 and will provide a general and simplified overview of the Control and Data flow of the architecture model for the client. Again, a number wrapped in parentheses implies the respective arrow in the figure above. The different steps in the process are described below in the (mostly) general case.

1. The user interacts with the html page which triggers an event that is sent to (4) the JavaScript interpreter.
2. The JavaScript interpreter calls (3) the JavaScript function which is mapped to the corresponding event.
3. The function might need to fetch or register some data on the server. So an asynchronous HTTP request (8, 9, 10) is made.
4. The server will respond (5, 4, 3) which will trigger the corresponding JavaScript event handler (10, 9, 8)
5. The event handler will do the required action, i.e. generate the required HTML and inject it into the html page.

## 3. Design Considerations

### 3.1 Assumptions and Dependencies

First off, before reading this section read the following in this design document (DD) and in the requirements document (RD):

- DD section 2.3 about the detailed architecture.
- RD section 2.1 about the intended users of the system
- RD section 2.6.1 and 2.6.2 about client and server side technologies.
- RD section 4.2 about the system's non-functional requirements.
- RD section 8 about system evolution. It covers a lot of assumptions about; the system, it's hardware and what in its functionality that is likely to be changed due to user needs.

#### 3.1.1 Related software

The server side of the system is built in JEE and therefore depends on a JVM to be available on the server in order to run the system. The server side of the system also depends on the existence of a database and a web server.

The client side of the system is built through a collection of ECMAScripts (mostly JavaScript but at least one ActionScript) and that depends on the existence of a web browser capable of running these scripts. The client side of the system also depends on the web browser being compliant to other web standards such as HTML, CSS and DOM.

The system is created based on the assumption that the dependencies above are matched.

The system will be tested with the following software:

- Server side
  - OS: Debian GNU/Linux with kernel 2.6.18-5-686
  - JVM: JVM 1.5.0\_14
  - database: PostgreSQL 8.1.11
  - web server: Apache Tomcat 5.5.20
- Client side
  - web browser: different versions of Mozilla Firefox with Flash 9 support

#### 3.1.2 Related hardware

The system is dependent on internet access.

It is likely that the server side of the system will have a higher demand of computational power, memory and bandwidth in the case of the system being widely used.

The system will be tested with the following hardware:

- Server side
  - internet access: 10 Mbit
  - computational power: 900 MHZ Intel Pentium III CPU
  - memory:

- primary: 256 MB RAM
  - secondary: 40 GB HD
- Client side
  - random computers with the appropriate (see previous section) web browser support

### **3.1.3 End-user characteristics**

The users of the system are supposedly young (10 to 30 years of age) and are therefore assumed to be up to date with current technologies. They are assumed to know their way around a standard web site and should therefore be able to navigate our system.

### **3.1.4 Possible and/or probable changes in functionality**

After some time using the system the users might get bored. Then changes that increase the game value to the system's users might be implemented.

These changes could involve:

- making the system available in alternative ways. For example via mobile devices.
- adding new functionality when painting pictures. For example new painting tools.
- introducing special events to the community. For example during Christmas all the topics will be related to Christmas.

## **3.2 General Constraints**

There are several limitations that constrain the software implementation of the system. Since it is web based and is used over the internet, it has to be small to make the game playable to people with slow internet connections. The game is to be played in the user's web browser, and that imposes other limitations, like having to use supported standards and protocols. The security and ability to verify a user is also very important in a web based system like this.

## 4. Graphical User Interface

### 4.1 Overview of the User Interface

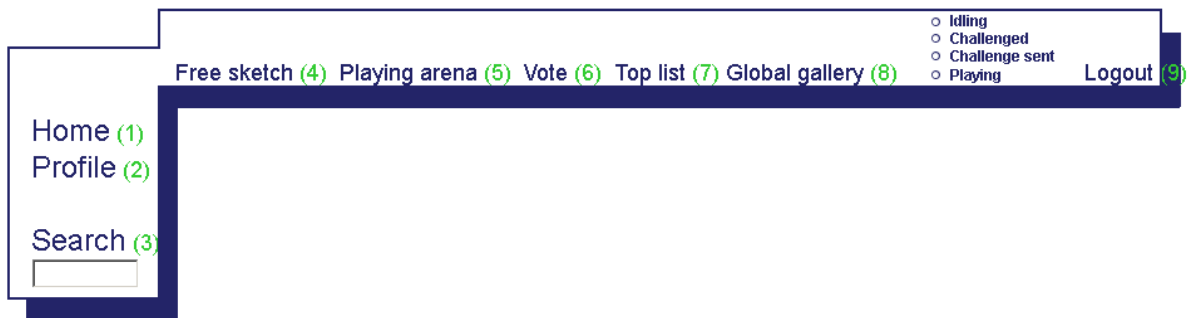
Please note that this part of the Design Document (DD) does most definitively not present the system's final layout (sizing, color, style). The figures should be viewed as guidelines when describing the functionality when a user interacts with the system through the system's different web pages that is explained below. The functionality described is however not a guideline.

Users will interact with the system via the user's web browser. In the user's web browser the system will be available through several web pages. These web pages will share some similar sections and have other sections that are unique for just that web page.



All of the system's web pages will include the menu shown in the figure above. This menu will provide the functionality to:

- (1) Direct the user to the "About" web pages which will let the user know who has created the system and which version of the system that is currently used.
- (2) Direct the user to the "Help" web pages which will teach the user how to use the system and help the user to get passed difficulties.



With the exception of the web pages concerning Login and Sign up (which are explained in DD section 4.2) all other web pages contained the menu shown in the figure above. This menu will provide the functionality to:

- (1) Direct the user to the "Home" web pages. For more information see 4.2.3
- (2) Direct the user to the "Profile" web pages. For more information see 4.2.5
- (3) Search for a user in the system by specifying it's username.
- (4) Direct the user to the "Free Sketch" web pages. For more information see 4.2.1
- (5) Direct the user to the "Playing Arena" web pages. For more information see 4.2.6
- (6) Direct the user to the "Vote" web pages. For more information see 4.2.10
- (7) Direct the user to the "Top list" web pages. For more information see 4.2.9
- (8) Direct the user to the "Global gallery" web pages. For more information see 4.2.2
- (9) Log out from the system.

## 4.2 Graphical user interface forms

### 4.2.1 Free Sketch

Functional requirements (FR) that are covered by everything written here are 4.1.4.1 – 4.1.5.1.



Please note that this is just a guideline of what the implementation will look like. What you see in the picture is on the left a few tools and colors that can be used while painting. To the right is the canvas upon which a user will draw pictures.

### 4.2.2 Global gallery

Date	Combatants	Winner	Looser	#Votes	Topic		
		P1				Filter	
2008-03-01	P1, P2	P1	P2	30	Clueless dog	-	
Name: P1 Total AP: 200 #Comp. 76 #Won comp. 58		Name: P2 Total AP: 143 #Comp. 27 #Won comp. 10		<i>Picture of P1</i>			<i>Picture of P2</i>
2008-02-10	P1, P3	P1	P3	20	Summer	+	

FR: 4.1.7.2

This is the global gallery page where all pictures ever made are posted for all users to view.

This page consists of a list of competitions. The list can be sorted with respect to any of the six subjects at the top and it can also be filtered by entering a string in one or more of the text fields

below the corresponding subject. For example, the list above is filtered to only show competitions where user P1 is the winner.

More information about the competition can be shown by clicking on the plus-sign at the right of each competition as shown in the picture above. There you can see the two pictures and some statistics about the two competitors.

### 4.2.3 Home

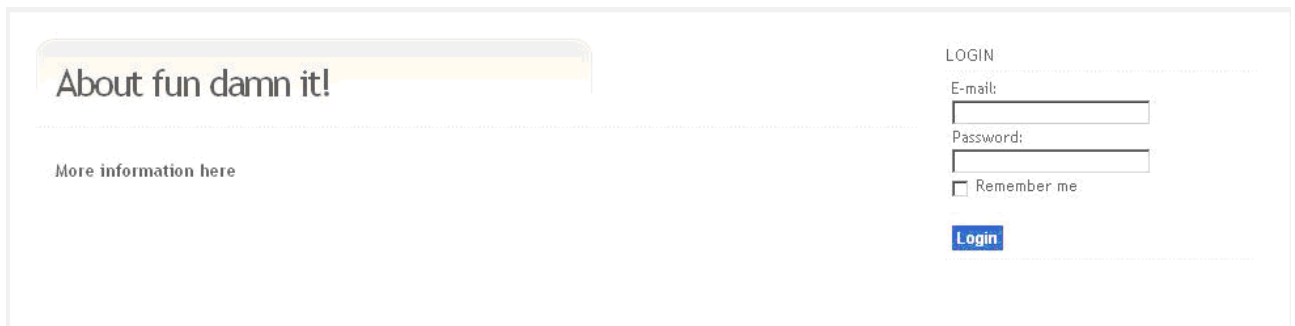
Welcome to Project Electra!

News	Upcoming Events
NewsDateLbl	EventDateLbl
NewsHeaderLbl	EventHeaderLbl
NewsMessageLbl	EventMessageLbl
NewsDateLbl	EventDateLbl
NewsHeaderLbl	EventHeaderLbl
NewsMessageLbl	EventMessageLbl

On the start page news and upcoming events are posted for the users to view. This page is only available to users that are logged in.

### 4.2.4 Login

The login page is quite self explanatory. All users are required to login to be able to use the system. As mentioned before this isn't the final design, but more of a guide line.



LOGIN

E-mail:

Password:

Remember me

The page covers the following functional requirements: 4.1.1.2

## 4.2.5 Profile

ProfilePictureArea	UsernameLbl	userStatsArea Total AP: totAPLbl
	PersonalMessageLbl	Participated in totCompLbl competitions Reserved AP: resAPLbl

<p><b>Guestbook</b></p> <hr/> <p>MessageDateLbl MessageLbl //SenderLbl</p> <hr/> <p>MessageDateLbl MessageLbl //SenderLbl</p> <p>New message:</p> <div style="border: 1px solid gray; height: 40px; width: 100%;"></div> <p style="text-align: center;">SubmitMessageBtn</p>	<p><b>Personal Gallery</b></p> <table border="1" style="width: 100%;"> <tr> <td style="width: 50%;">PictureArea</td> <td>Topic: topicLbl Won_LostLbl Opponent: opponentLbl</td> </tr> <tr> <td>PictureArea</td> <td>Topic: topicLbl Won_LostLbl Opponent: opponentLbl</td> </tr> </table>	PictureArea	Topic: topicLbl Won_LostLbl Opponent: opponentLbl	PictureArea	Topic: topicLbl Won_LostLbl Opponent: opponentLbl
PictureArea	Topic: topicLbl Won_LostLbl Opponent: opponentLbl				
PictureArea	Topic: topicLbl Won_LostLbl Opponent: opponentLbl				

Functional requirements that are covered by everything written here are 4.1.2.1-4.1.2.4. At the top we have the actual profile, where various information about the user is shown. Here we find game statistics, a user avatar, the user's name and a personal message written by the user. Next we have the user's guestbook, where the latest messages written to the user are displayed. Here we also have a form where other users can post a new message into the guestbook. At the right we have the personal gallery where the user's latest pictures are displayed along with information about the battle, e.g. if the user won or lost, what topic they was supposed to draw and against who the user battled.



## 4.2.6 Playing arena

FR that are covered by everything written here are 4.1.3.1 – 4.1.3.2.



To the left we have the game options area. This is where a user will specify the game options that will be sent to another user upon challenge. Time limit is chosen from a few predetermined options presented in the tList. The bet (1-100 %) is written in the betTextField. In the anteLabel the actual AP that the user will bet is shown.

FR: 4.1.3.7, 4.1.6.2.1, 4.1.6.2.3

At the top is the signupBtn. To interact with the rest of the controls on this page you must click on this button.

FR: 4.1.3.3

In the middle is the chat room. This comprise 3 parts: chatArea, competitorsArea and chatInputTextField. In general, to post a message a user types the message into the chatInputField and then clicks on the chatPostBtn. The message should then pop up in the chatArea. In the competitorsArea all users that are signed up and not participating in a battle at the moment are shown.

FR: 4.1.3.4 - 4.1.3.5

The competitorsArea also has a second use that are intertwined with the compInfoArea at the right. When a username in the competitorsArea is selected the compInfoArea will change.

The compInfoArea is used to present statistics about the selected user. These labels will be changed when a new username is selected in the competitorsArea:

totAPLbl – the total AP that the user currently has.

nCompLbl – the total number of competitions the user has participated in.

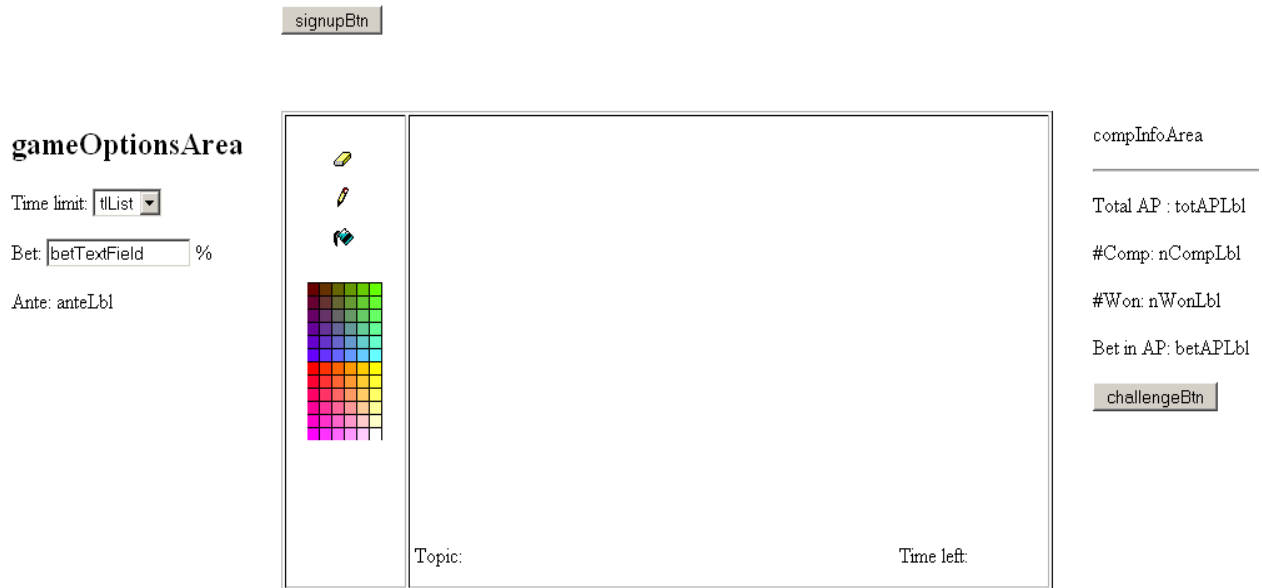
nWonLbl – the number of won competitions by the user.

The compInfoArea also has a button challengeBtn. When a user clicks on this the selected user will receive a challenge invitation using the selected battle options.

FR: 4.1.3.6 – 4.1.3.7

When a battle is commenced the chat room is replaced by a flash application that as depicted below (Note: just a guideline).

FR: 4.1.4.1 – 4.1.4.1, 4.1.6.3.1



## 4.2.7 Search

### Advanced search

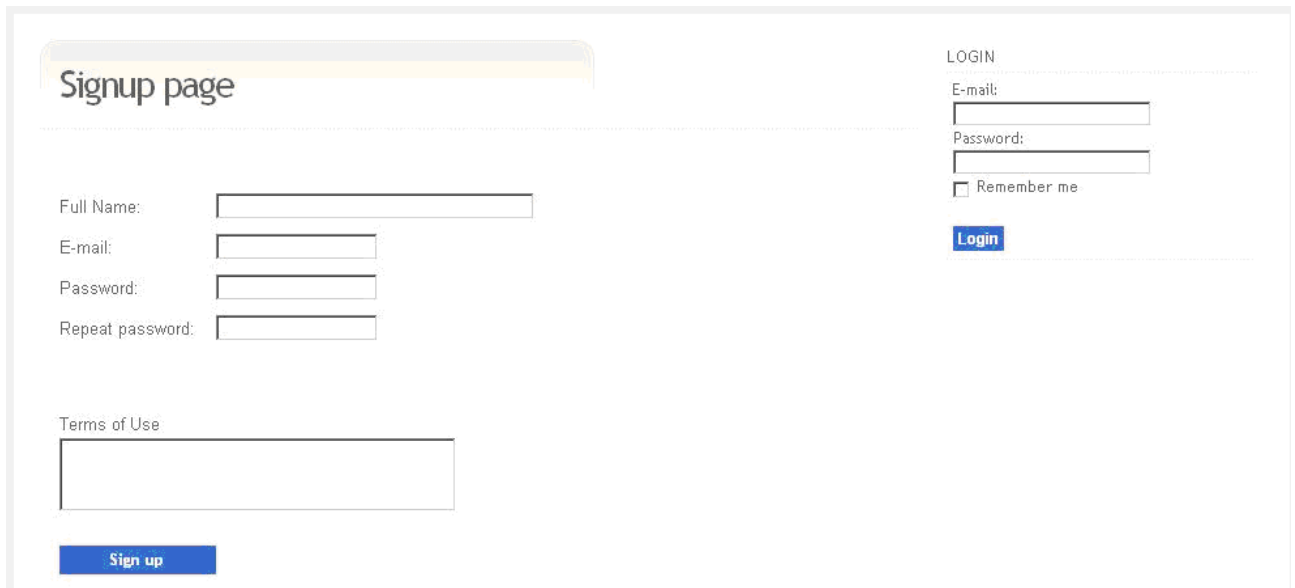
The image shows an "Advanced search" form. It has a title "Options" in bold. Below the title are three search criteria: "Search by name (1)" with an input field, "Number of competitions (2)" with a greater-than sign (>) and an input field, and "Total AP (3)" with a greater-than sign (>) and an input field. Below these are "Won competitions (4)" with a greater-than sign (>) and an input field, and a "Search (5)" button. Below the form is a table with a header row containing "(6)", "Name", a downward arrow (v), "Competitions", "Won", and "AP".

The web page shown in the figure above covers the functional requirement 4.1.7.3 and is used when searching for other users in the system.

- (1) Search for other users with regards for their username. Takes characters as input.
- (2) Search for other users with regards for their number of participated competitions. Any of the comparators “>, ≥, ≤ and <” can be used. Takes digits as input.
- (3) Search for other users with regards for AP. Any of the comparators “>, ≥, ≤ and <” can be used. Takes digits as input.
- (4) Search for other users with regards for their number of won competitions. Any of the comparators “>, ≥, ≤ and <” can be used. Takes digits as input.
- (5) This button performs the search for user given the options specified by (1) – (4).
- (6) This is a list of the results from a search. The result can be ordered by Name, Participated competitions”, Won competitions or Number of AP.

## 4.2.8 Sign up

As mentioned before, a user needs to be logged in to use the system. But before logging in they need to create an account once. The page where a new user can register and create an account is the following.



The image shows a web page titled "Signup page". On the left side, there is a registration form with the following fields: "Full Name:" (a long text input), "E-mail:" (a text input), "Password:" (a text input), and "Repeat password:" (a text input). Below these fields is a "Terms of Use" section with a checkbox and a text input area. At the bottom of the form is a blue "Sign up" button. On the right side of the page, there is a "LOGIN" section with a dashed border. It contains an "E-mail:" label above a text input, a "Password:" label above another text input, a "Remember me" checkbox, and a blue "Login" button.

The page covers the following functional requirements: 4.1.1.1

## 4.2.9 Vote

Time left (DD:TT:MM)	Combatants	Topic	Golden vote	
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="checkbox"/>	<input type="button" value="Filter"/>
00:00:00	P1, P2	Clueless dog	★	-
<i>One of the pictures</i>  <input type="button" value="Vote"/>		<i>The other picture</i>  <input type="button" value="Vote"/>		
00:05:22	P1, P2	Clueless dog		-

FR: 4.1.6.4.1 – 4.1.5.4.8

The voting page is like the global gallery except that this page contains all competitions that are in the voting phase. When the user viewing the page has not yet voted for this competition and has not participated in the competition, the user will be able to vote. The user is shown the two pictures, without knowing which belongs to which user, and a vote-button below each picture.

00:05:22	P1, P2	Clueless dog		-
<b>Name:</b> P1 <b>Total AP:</b> 200 Picture of P1 <b>Votes:</b> 38%	<b>Name:</b> P2 <b>Total AP:</b> 167 Picture of P2 <b>Votes:</b> 62%			

When the user viewing the page already has voted in the competition or the user is one of the combatants, this will be shown when the user views the competition. The user now sees some statistics about the combatants and also the development of the competition.

## 5. Design Details

### 5.1 Class Responsibility Collaborator (CRC) cards<sup>6</sup>

#### 5.1.1 View

Class: GlobalGalleryJSP

<b>Responsibilities</b>	<b>Collaborators</b>
Generates the HTML for the global gallery.	PersonalGalleryManager

Class: PersonalGalleryJSP

<b>Responsibilities</b>	<b>Collaborators</b>
Generates the HTML for the personal gallery.	PersonalGalleryManager

Class: VoteJSP

<b>Responsibilities</b>	<b>Collaborators</b>
Retrieves and registers a user vote Generates HTML for voting page.	VoteManager

Class: SearchJSP

<b>Responsibilities</b>	<b>Collaborators</b>
Generates the HTML for the search result page.	SearchManager

Class: GuestBookJSP

<b>Responsibilities</b>	<b>Collaborators</b>
Generates HTML for the guestbook of a user.	GuestBookManager

Class: BattleArenaJSP

<b>Responsibilities</b>	<b>Collaborators</b>
Generates HTML along with the Flash component	None

Class: PlayingArenaJSP

<b>Responsibilities</b>	<b>Collaborators</b>
Generates HTML for the Playing arena Page.	None

Class: LoginJSP

<b>Responsibilities</b>	<b>Collaborators</b>
Generates the HTML to handle user login	AuthorityManager

---

<sup>6</sup> <http://c2.com/doc/oopsla89/paper.html>

Class: RegisterJSP

<b>Responsibilities</b>	<b>Collaborators</b>
Generates the HTML to handle user registration	RegisterManager

### 5.1.2 Controller

Class: PersonalGalleryManager

<b>Responsibilities</b>	<b>Collaborators</b>
Gathers and returns a list with information about different competition a user has entered.	CompetitionAgent

Class: PersonalGalleryManager

<b>Responsibilities</b>	<b>Collaborators</b>
Gathers and returns a list with information about different competition a user has entered.	CompetitionAgent

Class: VoteManager

<b>Responsibilities</b>	<b>Collaborators</b>
Retrieves a user vote, looks up how many real votes it corresponds to and registers it with the corresponding competition. Then gives the voter a predefined amount of AP for voting. Voting can only be done once for every competition and user.	APUtil CompetitionAgent

Class: SearchManager

<b>Responsibilities</b>	<b>Collaborators</b>
Searches for users based on some given parameters and returns a list with the result.	UserAgent

Class: GuestBookManager

<b>Responsibilities</b>	<b>Collaborators</b>
Returns a users guestbook posts based on a timestamp and count.	GuestBookAgent

Class: CompetitionAgent

<b>Responsibilities</b>	<b>Collaborators</b>
Registers a vote from a user with the corresponding competition. Looks up and returns the different competitions which a user is connected to in the database.	Competition

Class: UserAgent

---

<b>Responsibilities</b>	<b>Collaborators</b>
Creates a user in the database. Retrieves users from the database.	User

Class: GuestBookAgent

<b>Responsibilities</b>	<b>Collaborators</b>
Creates a guestbook post in the database. Retrieves guestbook posts from the database.	GuestBookPost

Class: ChallengeServlet

<b>Responsibilities</b>	<b>Collaborators</b>
Handle user signup in the playing arena. Handle user challenge another user	PlayingArenaManager

Class: ChatServlet

<b>Responsibilities</b>	<b>Collaborators</b>
Handle get and post chat messages	PlayingArenaManager

Class: PlayingArenaManager

<b>Responsibilities</b>	<b>Collaborators</b>
Get and post chat messages.	ChatManager
Handle challenges. Handle Playing arena sign up.	ChallengeManager

Class: ChatManager

<b>Responsibilities</b>	<b>Collaborators</b>
Get and Post ChatMessages	ChatMessage

Class: ChallengeManager

<b>Responsibilities</b>	<b>Collaborators</b>
Keep track of users in the Playing arena.	ChallengeManagerUserEntry.
Create competitions.	CompetitionAgent

Class: AuthorityManager

<b>Responsibilities</b>	<b>Collaborators</b>
Handles user authentication	UserAgent
Handles user rights, used by all of the interfaces outward to check if a session/user has the right to perform the requested action.	UserAgent

Class: RegisterManager

<b>Responsibilities</b>	<b>Collaborators</b>
-------------------------	----------------------

Handles user registration	UserAgent
---------------------------	-----------

### 5.1.3 Model

Class: Competition

Responsibilities	Collaborators
Holds information about a competition. Is an entity that is stored in the database.	Topic Combatant

Class: Topic

Responsibilities	Collaborators
Holds information about a topic. Is an entity that is stored in the database.	None

Class: Combatant

Responsibilities	Collaborators
Holds information about a combatant. Is an entity that is stored in the database.	Picture

Class: Picture

Responsibilities	Collaborators
Holds information about a picture. Is an entity that is stored in the database.	None

Class: Class

Responsibilities	Collaborators
Holds information about a user class. Is an entity that is stored in the database.	None

Class: GuestBookPost

Responsibilities	Collaborators
Holds information about a guest book post. Is an entity that is stored in the database.	User

Class: News

Responsibilities	Collaborators
Holds information about the system that is presented as news. Is an entity that is stored in the database.	User

Class: User

Responsibilities	Collaborators
Holds information about a user. Is an entity	UserProfile



that is stored in the database.	
---------------------------------	--

Class: UserProfile

<b>Responsibilities</b>	<b>Collaborators</b>
Holds information about a user's profile. Is an entity that is stored in the database.	None

Class: PlayingArenaUserManager

<b>Responsibilities</b>	<b>Collaborators</b>
Keep track of all ChallengeRequests	ChallengeRequest
Keep track of ChallengeMessages.	Message

Class: ChallengeRequest

<b>Responsibilities</b>	<b>Collaborators</b>
Holds information about a challenge.	PlayingArenaUserManager

Class: Message

<b>Responsibilities</b>	<b>Collaborators</b>
A base class for returning various messages to the user.	None

Class: MessageChallenge

<b>Responsibilities</b>	<b>Collaborators</b>
A sub class of Message, tells the user client application that the user has been challenged.	None

Class: MessageGoToArena

<b>Responsibilities</b>	<b>Collaborators</b>
A sub class of Message, tells the user client application to go to the battle arena.	None

Class: MessageChat

<b>Responsibilities</b>	<b>Collaborators</b>
A class for posting messages in the chat.	None

Class: MessageOK

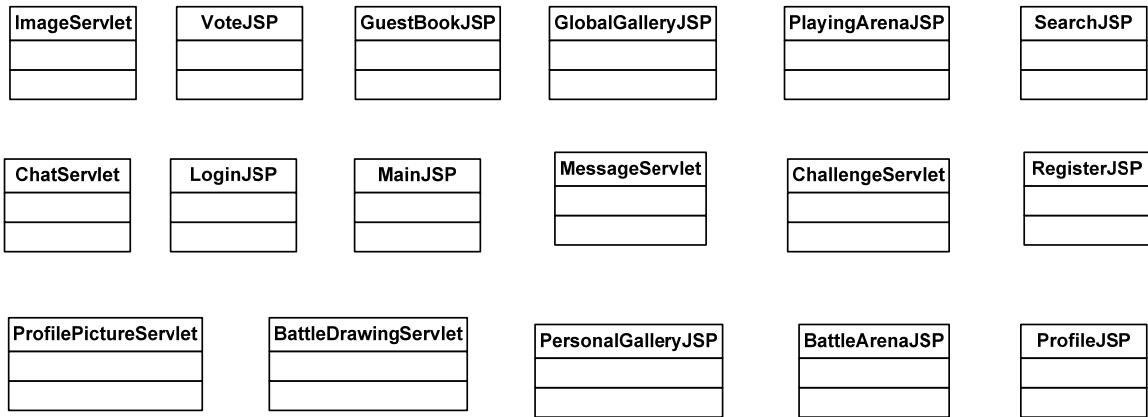
<b>Responsibilities</b>	<b>Collaborators</b>
A class that tells if something went ok.	None

Class: MessageError

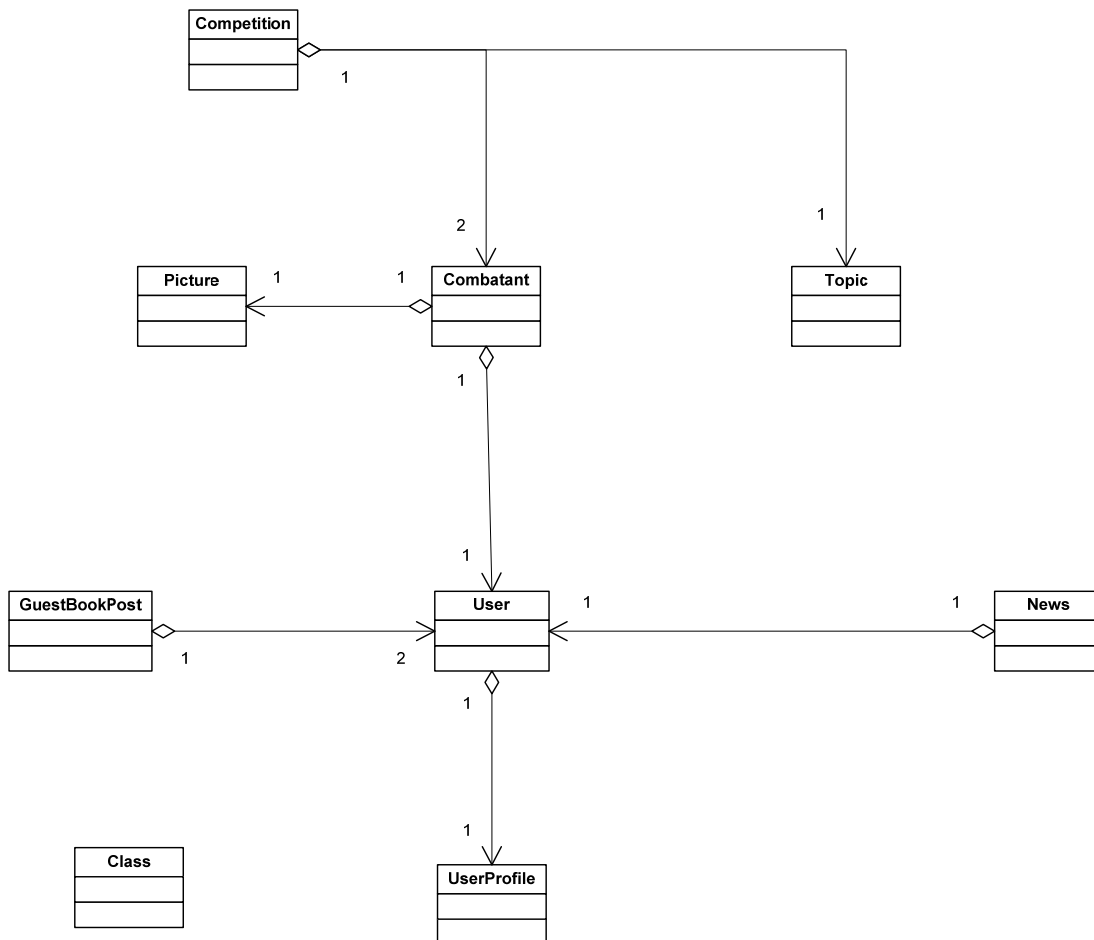
<b>Responsibilities</b>	<b>Collaborators</b>
A class that tells if something went wrong.	None

## 5.2 Class diagram<sup>7</sup>

### 5.2.1 View

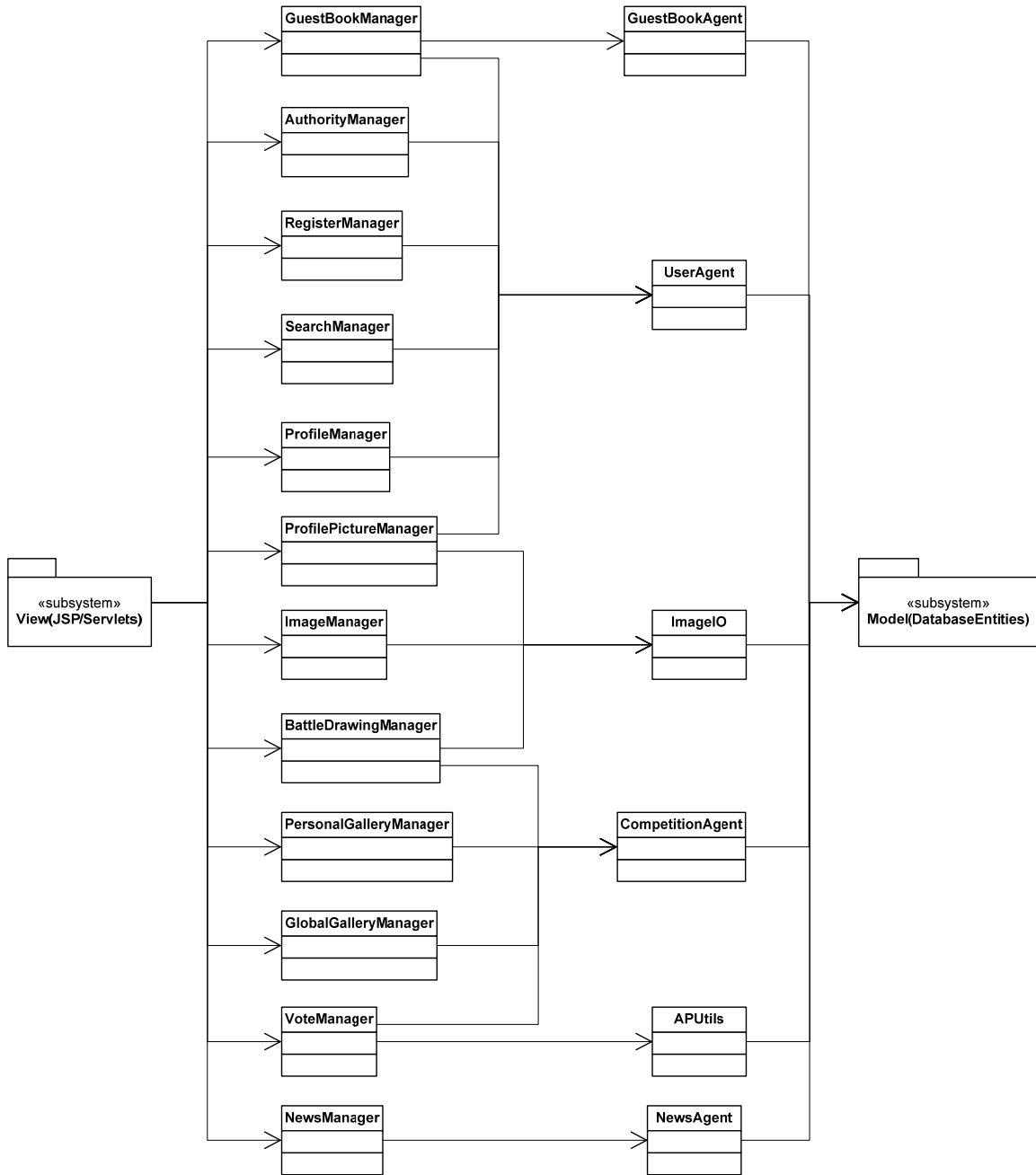


### 5.2.2 Model

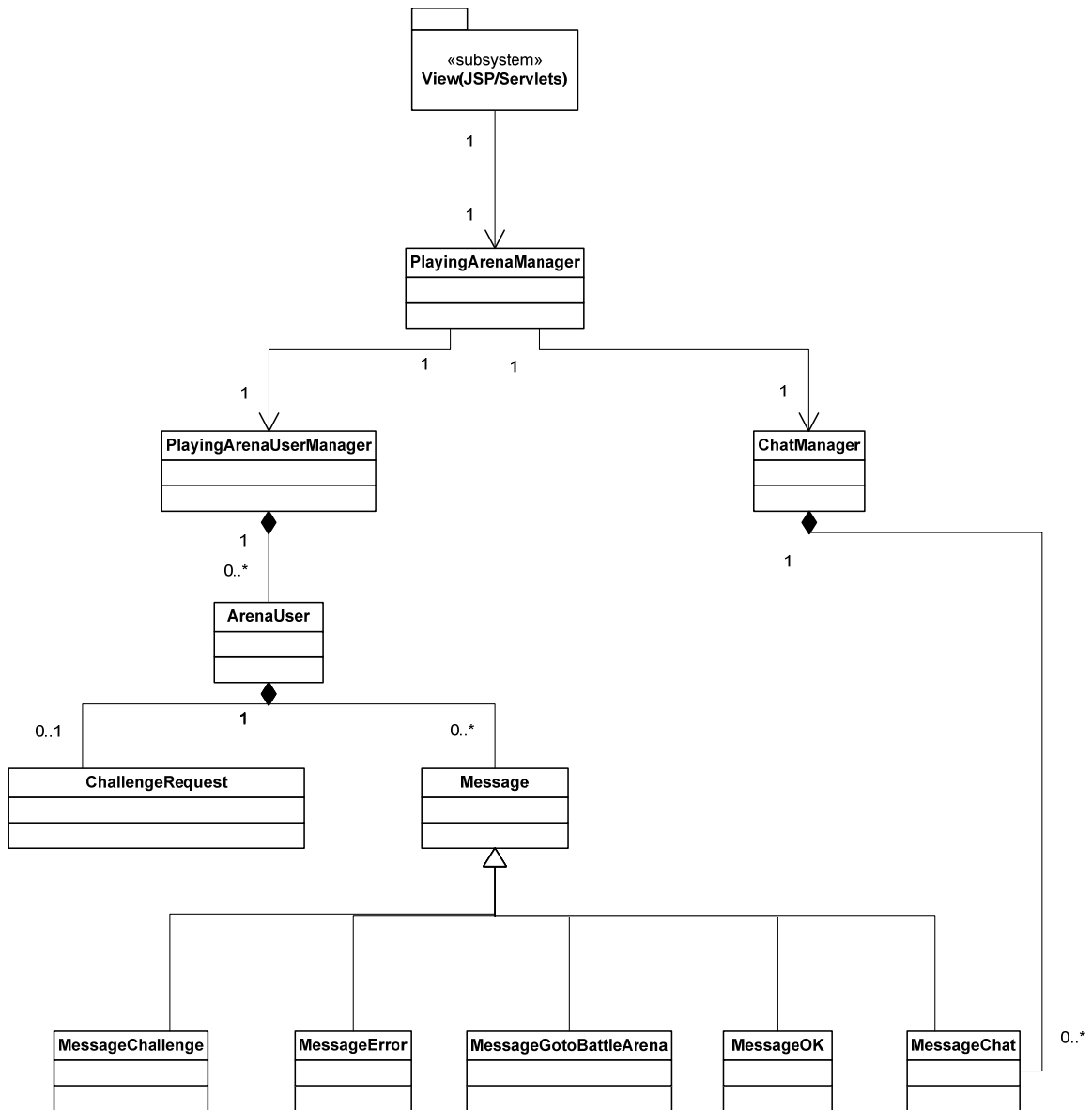


<sup>7</sup> <http://dn.codegear.com/article/31863#classdiagrams>

### 5.2.3 Controller

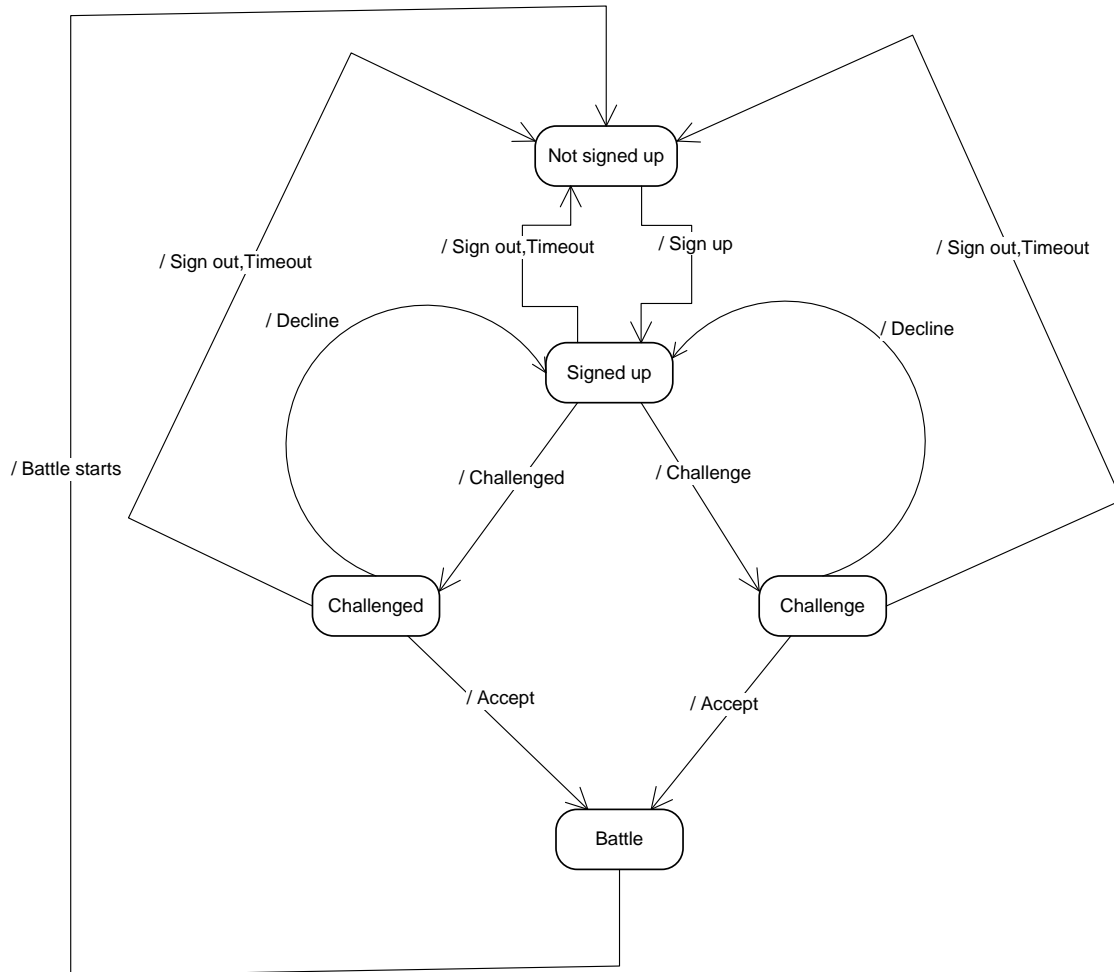


## 5.2.4 Playing Arena



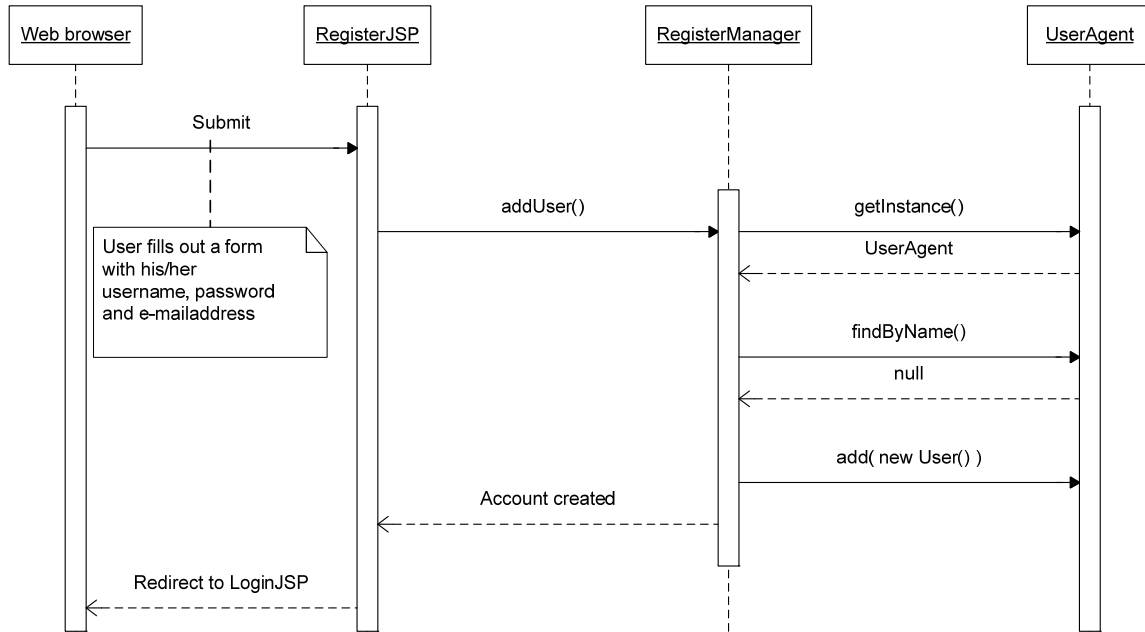
## 5.3 State charts

States of a user in the Playing Arena.



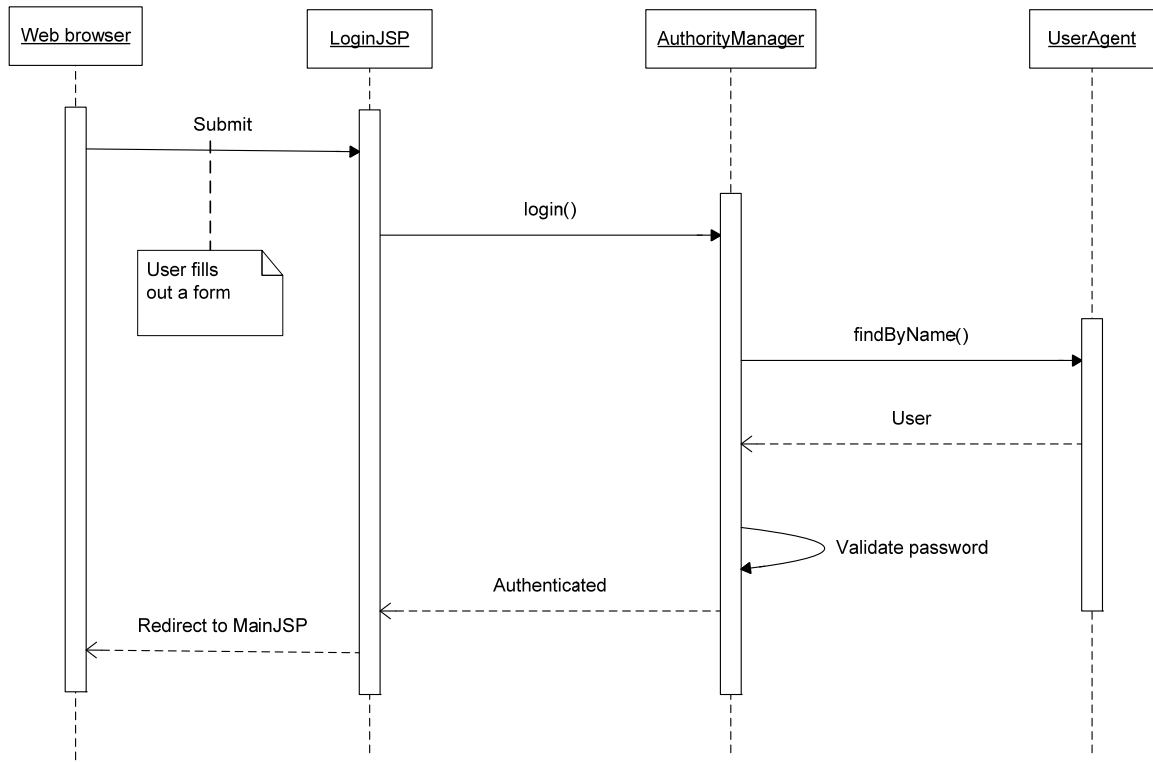
## 5.4 Interaction diagrams

### 5.4.1 Register



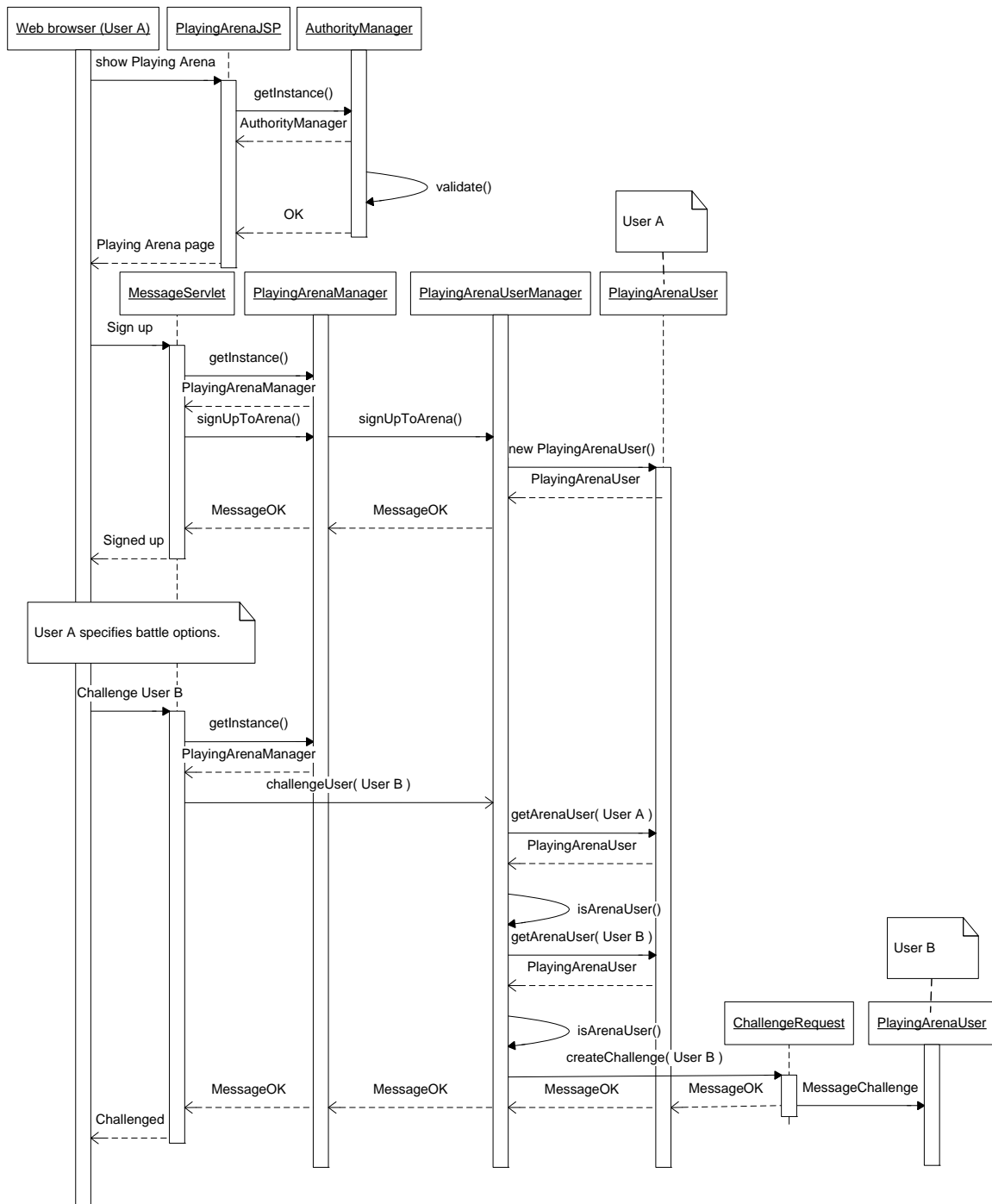
Reference to Use Case: 1

## 5.4.2 Login



Reference to Use Case: 2

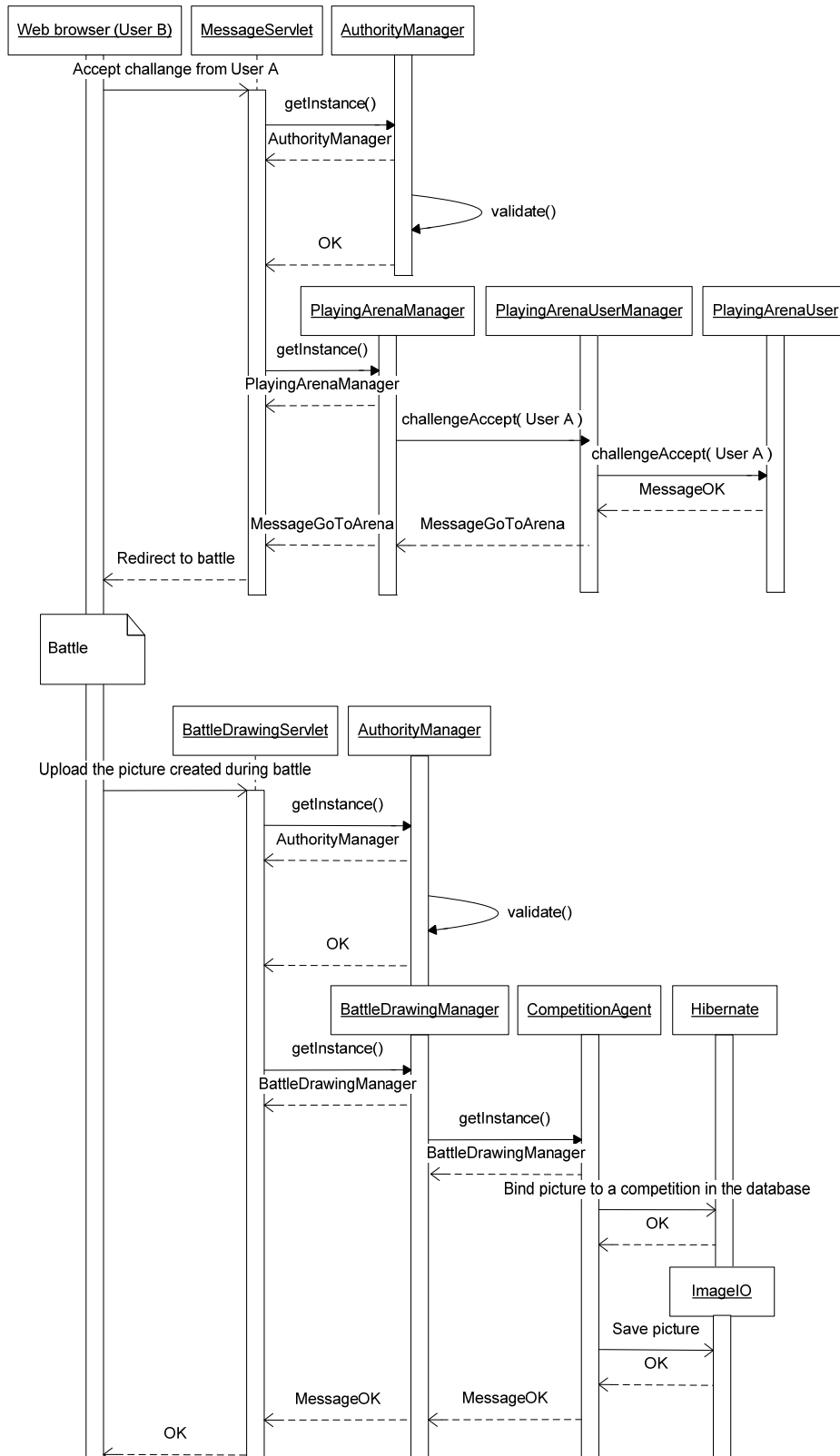
### 5.4.3 Challenge



Reference to Use Case: 3

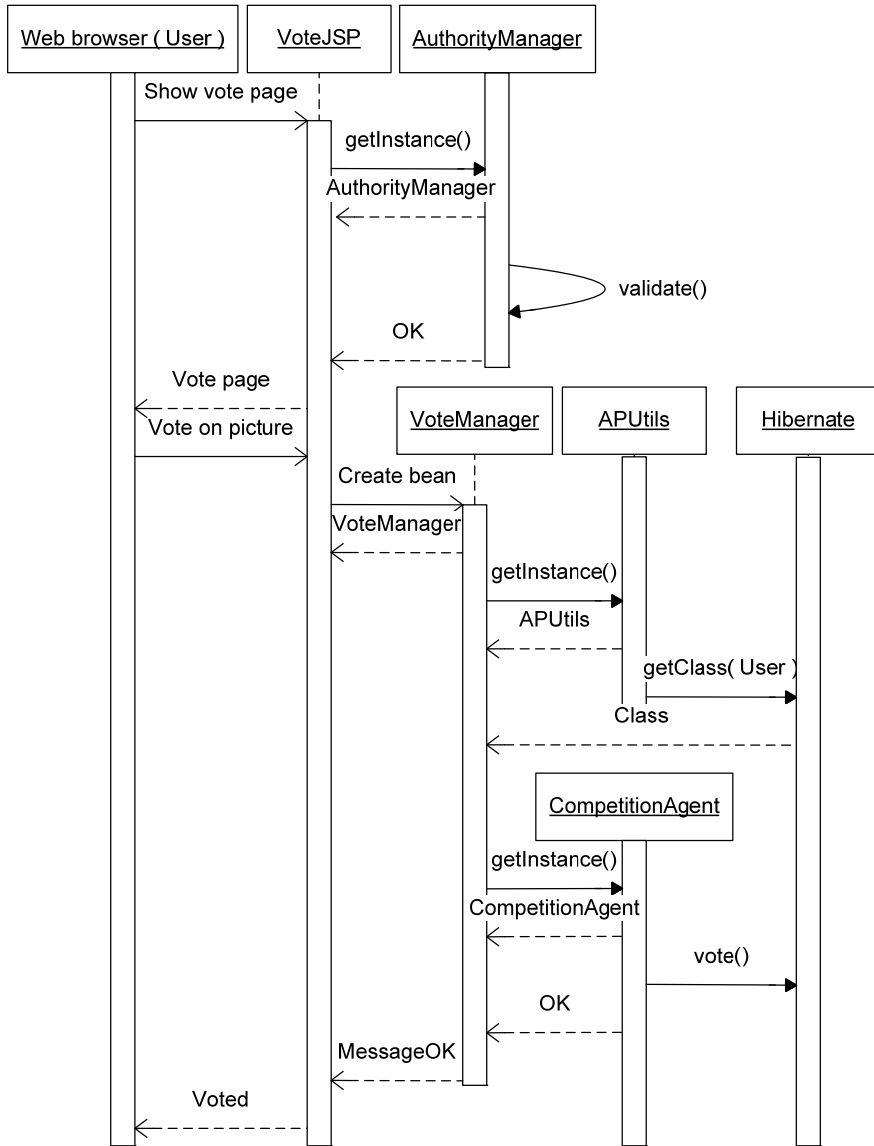


## 5.4.4 Battle



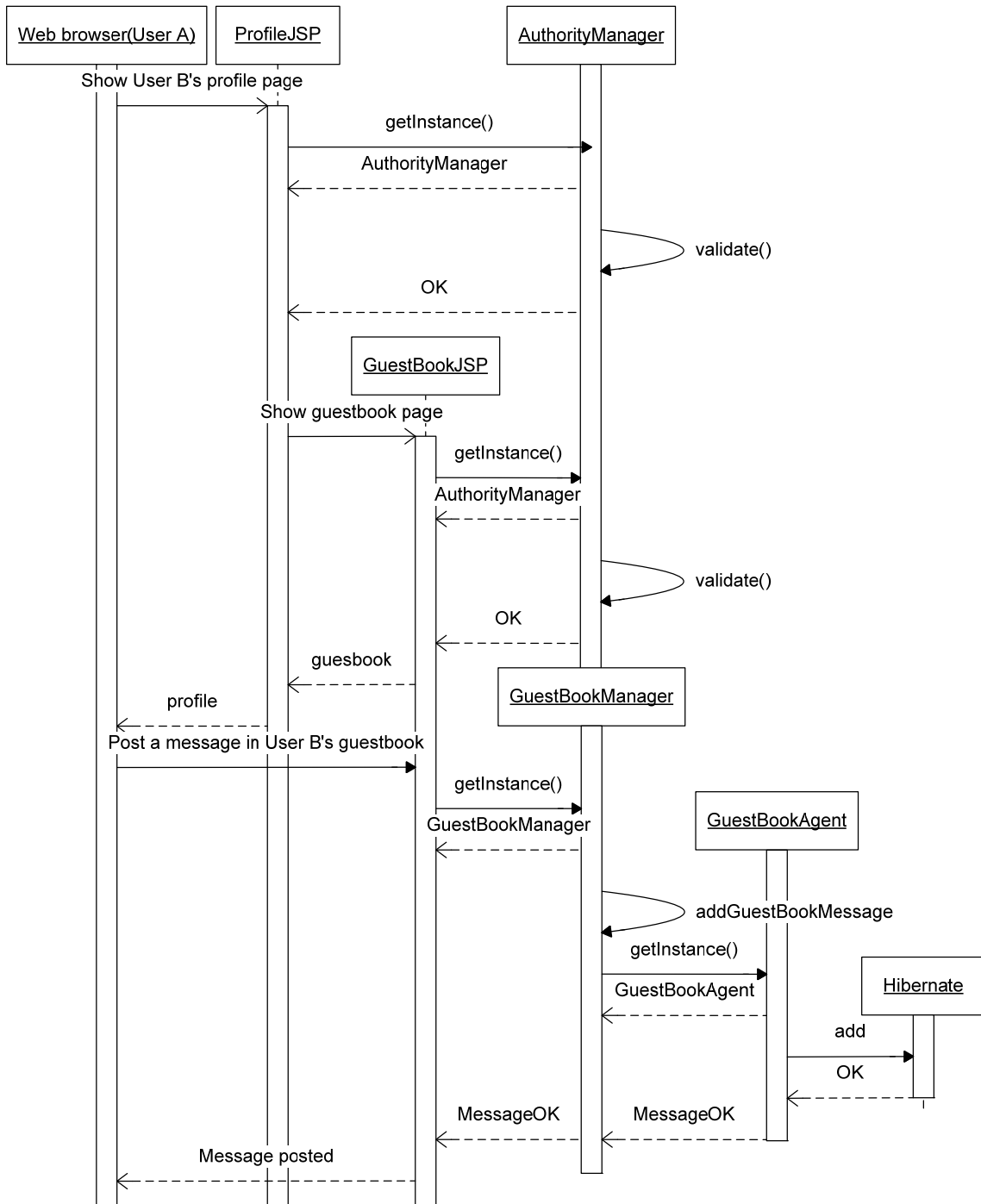
Reference to Use Case: 4

### 5.4.5 Vote



Reference to Use Case: 5

## 5.4.6 Add guestbook message



Reference to Use Case: 9

## 5.5 Detailed Design

### *APUtils Class Reference*

This is a utility class, responsible for making all types of calculations with the Artistic Points.

#### Public Member Functions

- void [updateClasses](#) ()  
*Updates the classes with new limits.*
  - Class [getClass](#) ([User](#) user)  
*Returns the class that the given user belongs to.*
  - int [getVoteWeight](#) ([User](#) user)  
*Returns the weight of the specified user's vote.*
  - [Combatant](#) [makeBet](#) ([User](#) bettingUser, int betPercent)  
*Calculates the user's bet according to the percent specified.*
  - void [transferAP](#) ([User](#) fromUser, [User](#) toUser, int amount)  
*Transfer 'amount' AP from 'fromUser' to 'toUser'.*
- 

#### Detailed Description

This is a utility class, responsible for making all types of calculations with the Artistic Points.

---

#### Member Function Documentation

##### **void updateClasses ()**

Updates the classes with new limits.

The limits depends on the AP of the user with the most AP.

##### **Pre condition>**

None

##### **Post condition>**

None

When the classes are modified they are saved in the database.

##### **Class getClass ([User](#) user)**

Returns the class that the given user belongs to.

##### **Pre condition>**

None

### Post condition>

None

#### ***Parameters:***

*user* An User object

#### ***Returns:***

The user's class.

### int getVoteWeight ([User](#) user)

Returns the weight of the specified user's vote.

### Pre condition>

None

### Post condition>

None

#### ***Parameters:***

*user*

#### ***Returns:***

The weight

### [Combatant](#) makeBet ([User](#) bettingUser, int betPercent)

Calculates the user's bet according to the percent specified.

Returns the same user with it's reserved AP increased or null if the user doesn't have enough free AP.

### Pre condition>

None

### Post condition>

None

#### ***Parameters:***

*bettingUser* The user that makes the bet

*betPercent* The percent of total AP.

#### ***Returns:***

The modified user.

### void transferAP ([User](#) fromUser, [User](#) toUser, int amount)

Transfer 'amount' AP from 'fromUser' to 'toUser'.

That is decreasing the fromUser's total AP by amount and increasing toUser's total AP by amount. Both user's reserved AP will be decreased by amount.

The method assumes that the transfer is possible i.e. that the fromUser's total AP is less or equal to amount.

**Pre condition>**

None

**Post condition>**

None

***Parameters:***

*fromUser*

*toUser*

*amount* The amount that shall be transfered

---

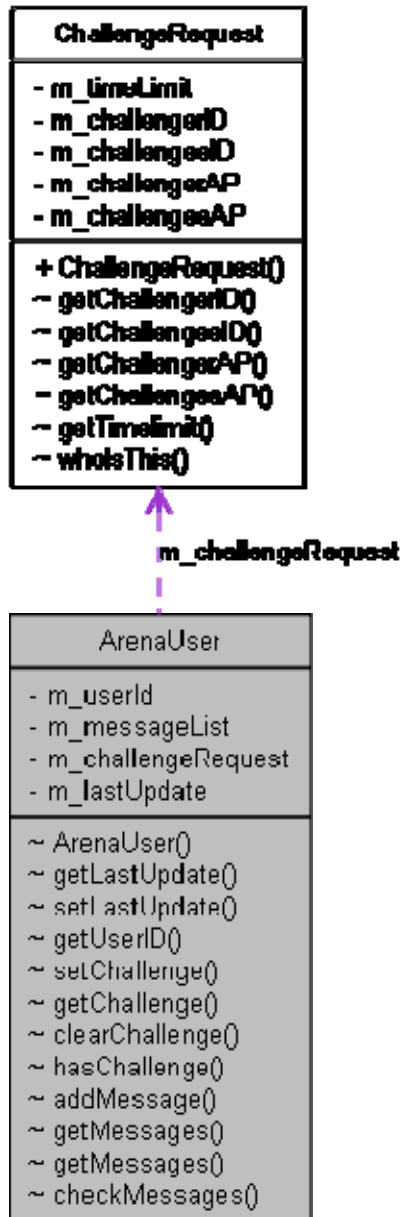
The documentation for this class was generated from the following file:

- [APUtils.java](#)

## ArenaUser Class Reference

The class represents an arena user.

Collaboration diagram for ArenaUser:



## Package Functions

- [ArenaUser](#) (long userId)  
*Constructor, initializes the object.*
- long [getLastUpdate](#) ()

Returns a timestamp that is the 'lastUpdate' property.

- void [setLastUpdate](#) ()  
Updates the last 'updated' field so that the arena user doesn't time out.
  - long [getUserID](#) ()  
Gets the user ID.
  - void [setChallenge](#) ([ChallengeRequest](#) c)  
Sets the arena users [challenge](#) object.
  - [ChallengeRequest](#) [getChallenge](#) ()  
Returns the users [challenge](#) request object.
  - void [clearChallenge](#) ()  
Clears the arena users [challenge](#) request object.
  - boolean [hasChallenge](#) ()  
Checks if the arena user has [challenge](#) request object.
  - void [addMessage](#) ([Message](#) m)  
Adds a [message](#) to the arena user [message](#) list.
  - List< [Message](#) > [getMessages](#) ()  
Returns all the arena user messages and clears the [message](#) list.
  - List< [Message](#) > [getMessages](#) ([Message.TYPE](#) type)  
Returns all the messages of a certain type from the [message](#) list and also removes them from the list.
  - boolean [checkMessages](#) ([Message.TYPE](#) type)  
Checks if the arena user has a [message](#) of a certain type.
- 

## Detailed Description

The class represents an arena user.

It contains the [message](#) list and a [challenge](#) request.

This class is not thread safe.

### **Author:**

Ali Mosavian

---

## Constructor & Destructor Documentation

### **[ArenaUser](#) (long *userId*) [package]**

Constructor, initializes the object.

### **Pre condition**

None

### **Post condition**

None



**Parameters:**

*userId* The user ID

---

**Member Function Documentation**

**long getLastUpdate () [package]**

Returns a timestamp that is the 'lastUpdate' property.

**Pre condition**

The object isn't being modified by another thread.

**Post condition**

None

**Returns:**

UTC timestamp

**void setLastUpdate () [package]**

Updates the last 'updated' field so that the arena user doesn't time out.

**Pre condition**

The object isn't being modified by another thread.

**Post condition**

'lastUpdate' is set to now.

Referenced by ChallengeManager.getUserMessages().

Here is the caller graph for this function:



**long getUserID () [package]**

Gets the user ID.

**Pre condition**

The object isn't being modified by another thread.

**Post condition**

None

**Returns:**

User ID

**void setChallenge ([ChallengeRequest c](#)) [package]**

Sets the arena users [challenge](#) object.

## Pre condition

The object isn't being modified by another thread.

## Post condition

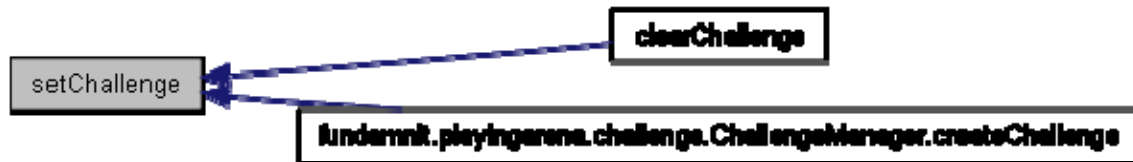
Challenge request object is 'c'

### Parameters:

*c* The [challenge](#) request object to set

Referenced by ArenaUser.clearChallenge(), and ChallengeManager.createChallenge().

Here is the caller graph for this function:



## [ChallengeRequest](#) getChallenge () [package]

Returns the users [challenge](#) request object.

## Pre condition

The object isn't being modified by another thread.

## Post condition

None

### Returns:

The object or null if there is none.

## void clearChallenge () [package]

Clears the arena users [challenge](#) request object.

## Pre condition

The object isn't being modified by another thread.

## Post condition

None

### Returns:

True if the user has a [challenge](#) request, false otherwise

References ArenaUser.setChallenge().

Here is the call graph for this function:



## boolean hasChallenge () [package]

Checks if the arena user has [challenge](#) request object.

### Pre condition

The object isn't being modified by another thread.

### Post condition

None

#### *Returns:*

True if the user has a [challenge](#) request, false otherwise  
Referenced by ChallengeManager.createChallenge().

Here is the caller graph for this function:



## void addMessage ([Message](#) m) [package]

Adds a [message](#) to the arena user [message](#) list.  
This method is not thread safe.

### Pre condition

The [message](#) list isn't being modified by another thread

### Post condition

The [message](#) has been added to the [message](#) list.

#### *Parameters:*

*m* The [message](#) to add to the arena user [message](#) list  
Referenced by ChallengeManager.createChallenge().

Here is the caller graph for this function:



## List<[Message](#)> getMessages () [package]

Returns all the arena user messages and clears the [message](#) list.  
This method is not thread safe.

### Pre condition

The [message](#) list isn't being modified by another thread

### Post condition

Message list is empty.

#### *Returns:*

A list of messages.

Referenced by ChallengeManager.getUserMessages().

Here is the caller graph for this function:



### List<[Message](#)> getMessages ([Message.TYPE](#) type) [package]

Returns all the messages of a certain type from the [message](#) list and also removes them from the list.

This method is not thread safe.

#### Pre condition

The [message](#) list isn't being modified by another thread

#### Post condition

All messages of the requested type have been removed from the [message](#) list.

#### Parameters:

*type* The [message](#) type to get

#### Returns:

A list of [message](#) of the requested type

### boolean checkMessages ([Message.TYPE](#) type) [package]

Checks if the arena user has a [message](#) of a certain type.

This method is not thread safe.

#### Pre condition

The [message](#) list isn't being modified by another thread

#### Post condition

None

#### Parameters:

*type* Message type

#### Returns:

True if any message(s) of that type were found

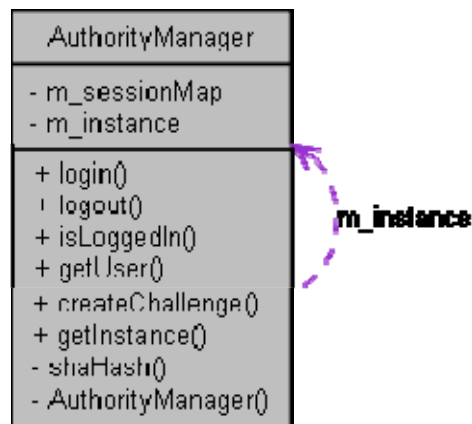
---

The documentation for this class was generated from the following file:

- [ArenaUser.java](#)

## AuthorityManager Class Reference

Collaboration diagram for AuthorityManager:



### Public Member Functions

- boolean [login](#) (String session, String email, String challenge, String passwordHash)  
*Authenticates the user.*
- void [logout](#) (String session)  
*Marks a user as logged out if the user is logged in.*
- boolean [isLoggedIn](#) (String session)  
*Checks if a user is logged in based on session ID.*
- [User](#) [getUser](#) (String session)  
*Returns a user object based on session ID, if the user is logged in.*
- String [createChallenge](#) ()  
*Creates a random 24 character challenge.*

### Static Public Member Functions

- static [AuthorityManager](#) [getInstance](#) ()  
*Returns the [AuthorityManager](#) instance.*

---

## Detailed Description

### Description

The authority manager handles authentication of users to the system by demanding that a user is logged in in order to use the system. This is validated every time a user tries to contact the system. The class is thread safe.

Per Almquist Ali Mosavian

## Member Function Documentation

### **boolean login (String *session*, String *email*, String *challenge*, String *passwordHash*)**

Authenticates the user.

The method assumes that the password which is passed to it has been appended with the challenge and hashed with SHA-1 and base64 encoded. The method will do the same and check if the hashed password matches. If so the session will be marked as logged in until the user logs out or the session expires.

#### **Pre condition**

None.

#### **Post condition**

If the authentication is successful, the users is marked as logged.

#### ***Parameters:***

*session* Session ID

*email* Users e-mail

*challenge* The password challenge

*passwordHash* The password + challenge hashes with SHA-1

#### ***Returns:***

True if the user is logged in or false if not

### **void logout (String *session*)**

Marks a user as logged out if the user is logged in.

#### **Pre condition**

User is logged in.

#### **Post condition**

User is marked as logged out.

#### ***Parameters:***

*session* Session ID of the user

### **boolean isLoggedIn (String *session*)**

Checks if a user is logged in based on session ID.

#### **Pre condition**

User is logged in.

#### **Post condition**

None

**Parameters:**

*session* Session ID of the user

**Returns:**

true if the user is logged in, false if not  
References `AuthorityManager.getUser()`.

Here is the call graph for this function:



**User getUser (String session)**

Returns a user object based on session ID, if the user is logged in.

**Pre condition**

User is logged in.

**Post condition**

None

**Parameters:**

*session* Session ID of the user

**Returns:**

User object if user is logged in, null otherwise  
Referenced by `AuthorityManager.isLoggedIn()`.

Here is the caller graph for this function:



**String createChallenge ()**

Creates a random 24 character challenge.

**Pre condition**

None

**Post condition**

None

**Returns:**

A 24 char long string containing the challenge.

**static AuthorityManager getInstance () [static]**

Returns the [AuthorityManager](#) instance.

It makes sure that there is only one UserAgent instance, if there is none, one is created.  
Implements the singleton design pattern.

**Pre condition(s)**

: None

**Post condition(s)**

: The singleton instance is created if required

***Returns:***

The singleton instance.

---

The documentation for this class was generated from the following file:

- [AuthorityManager.java](#)



## ***BattleDrawingManager Class Reference***

This class retrieves pictures from BattleDrawingServlet and binds them to the correct Competition and saves them in the file system.

### **Public Member Functions**

- void [postPicture](#) (Long *userId*, InputStream *picture*)  
*This method is invoked by BattleDrawingServlet, the userId is the id of the user that posted the picture.*
- 

### **Detailed Description**

This class retrieves pictures from BattleDrawingServlet and binds them to the correct Competition and saves them in the file system.

---

### **Member Function Documentation**

#### **void postPicture (Long *userId*, InputStream *picture*)**

This method is invoked by BattleDrawingServlet, the *userId* is the id of the user that posted the picture.

The method will create a Picture object and bind it to the correct Competition by using CompetitionAgent.

Then the picture will be saved in the filesystem with the help of [ImageIO](#).

#### ***Parameters:***

*userId*

*picture*

---

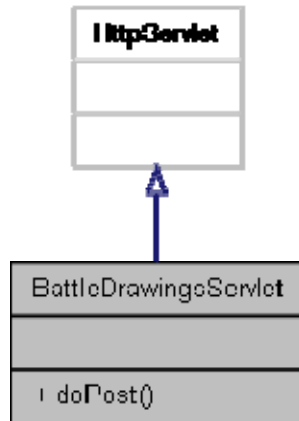
The documentation for this class was generated from the following file:

- [BattleDrawingManager.java](#)

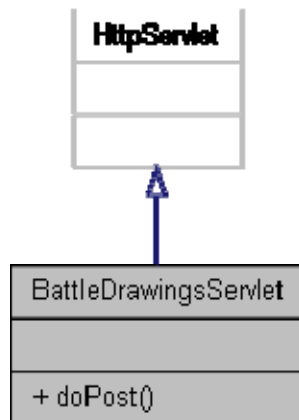
## *BattleDrawingsServlet Class Reference*

This class will receive requests from the Flash-component when the battle is over.

Inheritance diagram for BattleDrawingsServlet:



Collaboration diagram for BattleDrawingsServlet:



## Public Member Functions

- void [doPost](#) (HttpServletRequest req, HttpServletResponse resp)  
*The request parameter will contain binary data representing the picture send by the Flash-component.*

---

## Detailed Description

This class will receive requests from the Flash-component when the battle is over.

---

## Member Function Documentation

### **void doPost (HttpServletRequest *req*, HttpServletResponse *resp*)**

The request parameter will contain binary data representing the picture send by the Flash-component.

The method will obtain an InputStream to the binary data from the request parameter and the `userId` of the user that created the picture from the session.

The [BattleDrawingManager](#) is then responsible for saving the picture in the file system.

#### ***Parameters:***

*req* The Request parameter.

*resp* The Response parameter.

---

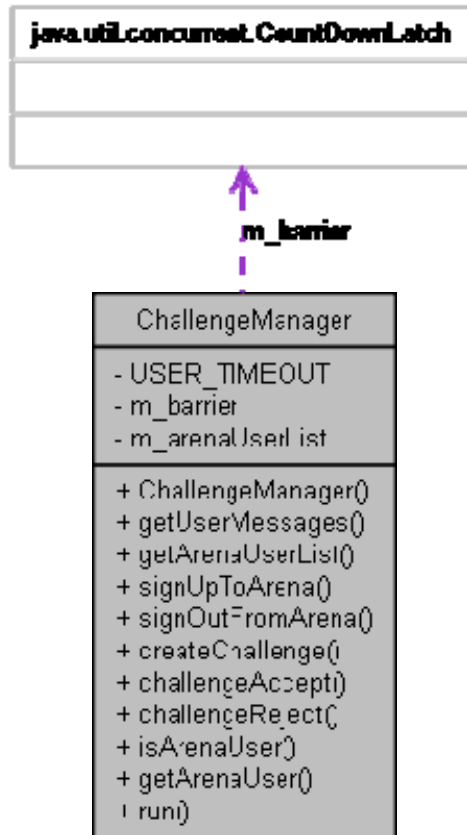
The documentation for this class was generated from the following file:

- [BattleDrawingsServlet.java](#)

## ChallengeManager Class Reference

This class handles all challenges and playing arena sign ups.

Collaboration diagram for ChallengeManager:



## Public Member Functions

- [ChallengeManager](#) (long userTimeout)  
*Constructor, initializes the ChallengeManager object.*
- List< [Message](#) > [getUserMessages](#) (long userID)  
*Returns a users messages if he/she is signed in to the playing arena and updates the last update field of the user.*
- Map< String, Long > [getArenaUserList](#) ()  
*Returns a sorted map of playing arena users.*
- [Message signUpToArena](#) (long userID)  
*Signs up a user to the playing arena.*
- [Message signOutFromArena](#) (long userID)  
*Signs out a user from the playing arena, prohibiting them from being challenged and challenging others.*
- [Message createChallenge](#) (long challengeeID, long challengerID, int challengeeAP, int challengerAP, int timelimit)  
*Creates a [challenge](#) request.*

- [Message challengeAccept](#) (int userID)  
*This method is called by a [challenge](#) when the user accepts a [challenge](#) request.*
  - [Message challengeReject](#) (int userID)  
*This method is called by challengee to reject a [challenge](#) request or by challenger to cancel [challenge](#) request.*
  - boolean [isArenaUser](#) (long userID)  
*Checks if the user is signed in to the playing arena.*
  - [ArenaUser getArenaUser](#) (long id)  
*Returns an ArenaUserEntry based on user id if the user is signed up in the arena, or null if he/she isn't.*
  - void [run](#) ()  
*This thread runs every 500 milliseconds and checks which users havent updated for USER\_TIMEOUT milliseconds and remove their challenges, and sign them out.*
- 

## Detailed Description

This class handles all challenges and playing arena sign ups.  
All methods are thread safe. Timed out users are signed out.

### **Author:**

Ali Mosavian

---

## Constructor & Destructor Documentation

### [ChallengeManager](#) (long *userTimeout*)

Constructor, initializes the ChallengerManager object.

### Pre condition

None

### Post condition

None

### **Parameters:**

*userTimeout* The number of milliseconds before a user should be considered timed out (>1000)

---

## Member Function Documentation

### List<[Message](#)> [getUserMessages](#) (long *userID*)

Returns a users messages if he/she is signed in to the playing arena and updates the last update field of the user.

If not null is returned. If there are no messages an empty list is returned.

## Pre condition

None

## Post condition

None

### *Parameters:*

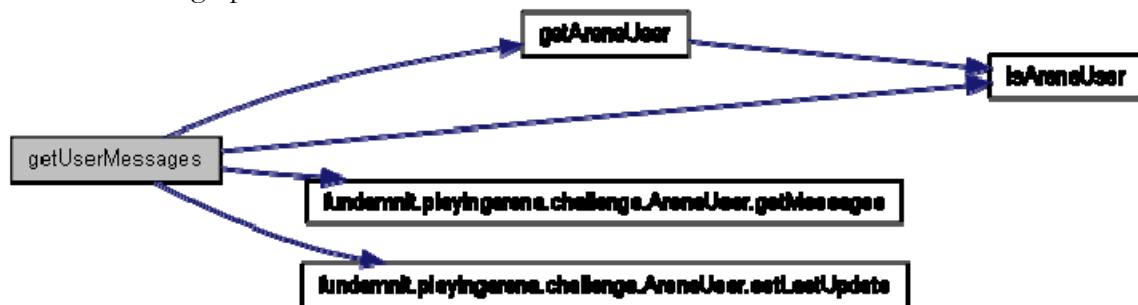
*userID* ID of user

### *Returns:*

null, empty list or list with messages.

References ChallengeManager.getArenaUser(), ArenaUser.getMessages(), ChallengeManager.isArenaUser(), and ArenaUser.setLastUpdate().

Here is the call graph for this function:



## Map<String, Long> getArenaUserList ()

Returns a sorted map of playing arena users.

## Pre condition

None

## Post condition

None

### *Returns:*

Sorted map with <user name, user id>

Referenced by `PlayingArenaManager.getArenaUserList()`.

Here is the caller graph for this function:



## [Message](#) signUpToArena (long *userID*)

Signs up a user to the playing arena.

Enabling them to be challenged and [challenge](#) other users. This method is thread safe.

## Pre condition

The user isn't already signed up.

## Post condition

User is signed up in the arena. User can be challenged.

### **Parameters:**

*userID* The user (id) to sign up/add to the playing arena

### **Returns:**

MessageOK on success or MessageError on failing.

## Message `signOutFromArena (long userID)`

Signs out a user from the playing arena, prohibiting them from being challenged and challenging others.

Any active challenges which hasn't been turned into an competition.

## Pre condition

User is signed up in the arena.

## Post condition

User is removed from the arena

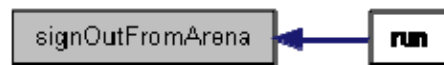
### **Parameters:**

*userID*

### **Returns:**

MessageOK or MessageError  
Referenced by ChallengeManager.run().

Here is the caller graph for this function:



## Message `createChallenge (long challengeeID, long challengerID, int challengeeAP, int challengerAP, int timelimit)`

Creates a [challenge](#) request.

But before that it makes sure that both users are signed in to the playing arena (not the site!) and that neither of them currently have active challenges. This method is thread safe.

## Pre condition

Both users are signed up in the playing arena. Neither have on going [challenge](#) request or competition.

## Post condition

Either no change, or the same [challenge](#) request have been added to both users and a MessageChallenge has been added to the challengees [message](#) list.

### **Parameters:**

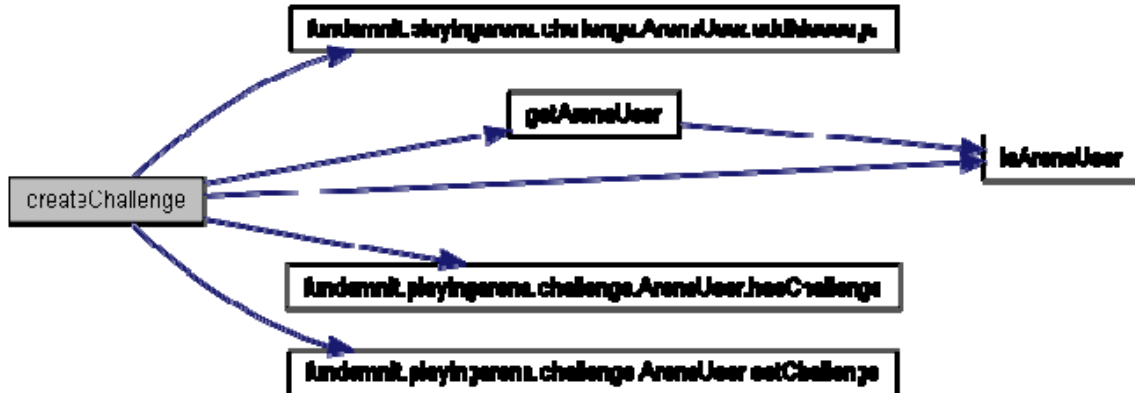
*challengeeID* The id of the user being challenged

*challengee.AP*  
*challengerID* The id of the user who is challenging  
*challenger.AP*  
*timelimit*

**Returns:**

MessageOK on success or MessageError on failing.  
 References ArenaUser.addMessage(), ChallengeManager.getArenaUser(), ArenaUser.hasChallenge(), ChallengeManager.isArenaUser(), and ArenaUser.setChallenge().

Here is the call graph for this function:



**Message challengeAccept (int userID)**

This method is called by a [challenge](#) when the user accepts a [challenge](#) request.

**Pre condition**

The user has been challenged. The challenger is still signed in to the playing arena.

**Post condition**

A competition is created through competition [agent](#). The [challenge](#) is cleared for both users, a [message](#) is returned that the challengee should go to the battle arena. A [message](#) is left for the challenger to go to the battle arena.

**Parameters:**

*userID* The id of the challengee

**Returns:**

MessageGoToBattleArena or MessageError

**Message challengeReject (int userID)**

This method is called by challengee to reject a [challenge](#) request or by challenger to cancel [challenge](#) request.

The method is thread safe.

**Pre condition**

The user has been challenged/has challenged someone.



## Post condition

The [challenge](#) is removed from both users, a corresponding [message](#) is left for the other user if required.

### **Parameters:**

*userID* The id of the user

### **Returns:**

MessageOK or MessageError on failure

## boolean isArenaUser (long *userID*)

Checks if the user is signed in to the playing arena.

## Pre condition

None

## Post condition

None

### **Parameters:**

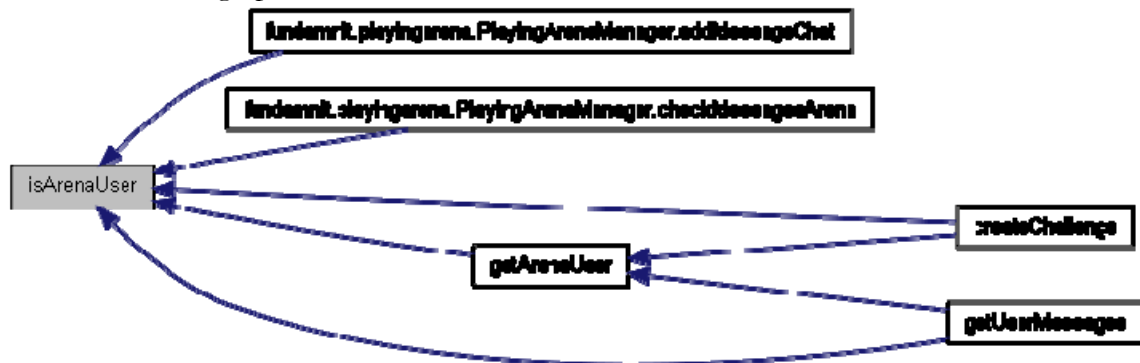
*userID* The id of the user to check

### **Returns:**

true or false

Referenced by `PlayingArenaManager.addMessageChat()`, `PlayingArenaManager.checkMessagesArena()`, `ChallengeManager.createChallenge()`, `ChallengeManager.getArenaUser()`, and `ChallengeManager.getUserMessages()`.

Here is the caller graph for this function:



## [ArenaUser](#) getArenaUser (long *id*)

Returns an `ArenaUserEntry` based on user id if the user is signed up in the arena, or null if he/she isn't.

This method is thread safe and is meant to be used within the package only.

## Pre condition

The user list isn't being modified by another thread.

## Post condition

None

### *Parameters:*

*id* The ID of the user to return

### *Returns:*

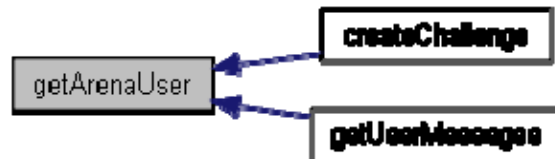
Instance of Arena user entry or null  
References ChallengeManager.isArenaUser().

Referenced by ChallengeManager.createChallenge(), and ChallengeManager.getUserMessages().

Here is the call graph for this function:



Here is the caller graph for this function:



## void run ()

This thread runs every 500 milliseconds and checks which users havent updated for USER\_TIMEOUT milliseconds and remove their challenges, and sign them out.

## Post condition

None

## Post condition

None

References ChallengeManager.signOutFromArena().

Here is the call graph for this function:



---

The documentation for this class was generated from the following file:

- [ChallengeManager.java](#)

## *ChallengeRequest Class Reference*

### Public Member Functions

- [ChallengeRequest](#) (long challengerID, int challengerAP, long challengeeID, int challengeeAP, int timelimit)  
*Initializes the object.*

### Package Types

- enum [TYPE](#)

### Package Functions

- long [getChallengerID](#) ()  
*Returns challenger user ID.*
- long [getChallengeeID](#) ()  
*Returns Challengee ID.*
- int [getChallengerAP](#) ()  
*Returns challenger betted AP.*
- int [getChallengeeAP](#) ()  
*Returns challengee betted AP.*
- int [getTimelimit](#) ()  
*Returns battle timelimit in seconds.*
- [TYPE whoIsThis](#) (long userID)  
*Checks if the passes user ID is challenger, challengee or neither.*

---

## Detailed Description

### *Author:*

Ali Mosavian

---

## Member Enumeration Documentation

enum [TYPE](#) [package]

Enumerator:

*CHAT*  
*TEXT*  
*OK*

*FAIL*  
*ERROR*  
*BEEN\_CHALLENGED*  
*CHALLENGE\_CANCELED*  
*ACCEPT*  
*GO\_TO\_BATTLE\_ARENA*  
*NEITHER*  
*CHALLENGER*  
*CHALLENGEE*

---

## Constructor & Destructor Documentation

[ChallengeRequest](#) (long *challengerID*, int *challengerAP*, long *challengeeID*, int *challengeeAP*, int *timelimit*)

Initializes the object.

### Post condition

None

### Post condition

None

### *Parameters:*

*challengerID* User ID of the challenger  
*challengerAP* Betted AP of challenger  
*challengeeID* User ID of the challengee  
*challengeeAP* Betted AP of challengee  
*timelimit* Battle timelimit in seconds

---

## Member Function Documentation

**long getChallengerID () [package]**

Returns challenger user ID.

### Post condition

None

### Post condition

None

### *Returns:*

Challenger ID

**long getChallengeeID () [package]**

Returns Challengee ID.

**Post condition**

None

**Post condition**

None

**Returns:**

Challengee ID.

**int getChallengerAP () [package]**

Returns challenger betted AP.

**Post condition**

None

**Post condition**

None

**Returns:**

AP

**int getChallengeeAP () [package]**

Returns challengee betted AP.

**Post condition**

None

**Post condition**

None

**Returns:**

AP

**int getTimelimit () [package]**

Returns battle timelimit in seconds.

**Post condition**

None

**Post condition**

None

**Returns:**

Battle timelimit

## **TYPE whoIsThis (long *userID*) [package]**

Checks if the passes user ID is challenger, challengee or neither.

### **Post condition**

None

### **Post condition**

None

### ***Returns:***

[TYPE.CHALLENGER](#), TYPE.CHALLENGE, [TYPE.NEITHER](#)

---

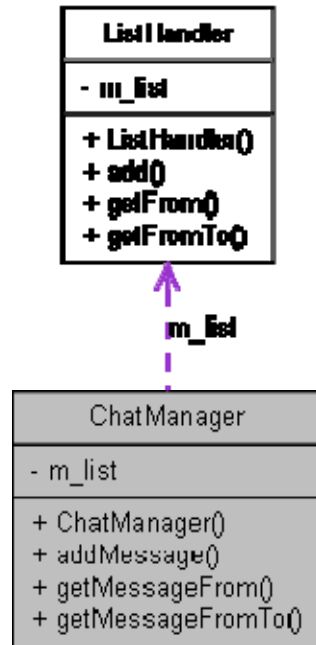
The documentation for this class was generated from the following file:

- [ChallengeRequest.java](#)

## ChatManager Class Reference

This class is the main playing arena [chat](#) component.

Collaboration diagram for ChatManager:



## Public Member Functions

- [ChatManager](#) (int size)  
*Initializes the [chat](#) manager.*
- void [addMessage](#) (int userID, String userName, String message)  
*Adds a [message](#) to the [chat message](#) list, replacing the last one if the max list size has been reached.*
- List< [MessageChat](#) > [getMessageFrom](#) (long timestamp, int count)  
*This method will return a maximum of 'count' number of messages from the passed timestamp and forward.*
- List< [MessageChat](#) > [getMessageFromTo](#) (long first, long last)  
*Will return all messages with are with a given timestamp range. The Method is thread safe.*

---

## Detailed Description

This class is the main playing arena [chat](#) component.

All [chat](#) messages go through this class. All methods are synchronized and are thread safe.

### **Author:**

Ali Mosavian

---

## Constructor & Destructor Documentation

### [ChatManager](#) (int *size*)

Initializes the [chat](#) manager.

#### Pre condition

$0 < \text{size} < 1000$

#### Post condition

Object initialized

#### *Parameters:*

*size* The maximum number of messages to hold

---

## Member Function Documentation

### void [addMessage](#) (int *userID*, String *userName*, String *message*)

Adds a [message](#) to the [chat message](#) list, replacing the last one if the max list size has been reached.

The method is thread safe.

#### Pre condition

Valid userID and userName

#### Post condition

None

#### *Parameters:*

*userID* User ID of [message](#) author

*userName* (nick) Name of the author

*message* The [message](#) text

References [ListHandler.add\(\)](#).

Referenced by [PlayingArenaManager.addMessageChat\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



### List<[MessageChat](#)> [getMessageFrom](#) (long *timestamp*, int *count*)

This method will return a maximum of 'count' number of messages from the passed timestamp and forward.



The method is thread safe.

## Pre condition

None

## Post condition

None

### **Parameters:**

*timestamp* The first messages timestamp will be either equal to this parameters or greater if an exact match isn't found.

*count* Number of messages to retrieve from the first [message](#). This is the maximum number of messages that will be returned. But the actual count might be less depending on how many are available from that point on. Count will be clipped to to the maximum list size.

### **Returns:**

A list that contains from 0 to count ChatMessage objects.

References ListHandler.getFrom().

Referenced by PlayingArenaManager.getMessagesChat().

Here is the call graph for this function:



Here is the caller graph for this function:



## List<[MessageChat](#)> getMessageFromTo (long *first*, long *last*)

Will return all messages with are with a given timestamp range.The Method is thread safe.

## Pre condition

Parameter 'first' > 'last'

## Post condition

None

### **Parameters:**

*first* The first messages timestamp will be either equal to this parameters or greater if an exact match isn't found.

*last* The last [message](#) timestamp, the actual last [message](#) returned will be either equal to or less then this.

### **Returns:**

A list containing a minimum of zero to a maximum as large as the circular list size messages.

References ListHandler.getFromTo().

Referenced by PlayingArenaManager.getMessagesChat().

Here is the call graph for this function:



Here is the caller graph for this function:



---

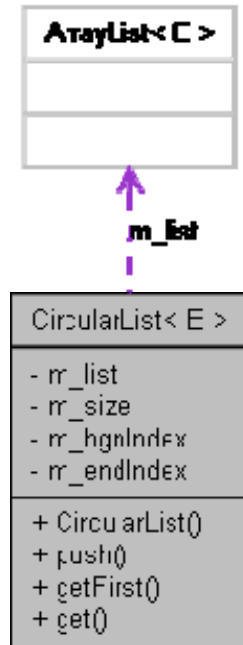
The documentation for this class was generated from the following file:

- [ChatManager.java](#)

## *CircularList< E > Class Reference*

A circular list data structure.

Collaboration diagram for CircularList< E >:



## Public Member Functions

- [CircularList](#) (int size)  
*The list constructor.*
- void [push](#) (E item)  
*This method will insert an item at the beginning of the list.*
- E [getFirst](#) ()  
*Returns the first item of the list.*
- E [get](#) (int index)  
*Returns item 'index'.*

---

## Detailed Description

A circular list data structure.

This class is not thread safe.

### **Author:**

Ali Mosavian

---

## Member Function Documentation

### CircularList (int *size*)

The list constructor.

#### Pre condition

None

#### Post condition

None

#### *Parameters:*

*size* The circular list size

### void push (E *item*)

This method will insert an item at the beginning of the list.

If the list has reached the maximum size the last item will be replaced and the last item will become the item before it.

#### Pre condition

None

#### Post condition

First item of list is 'item', last item might have been replaced.

#### *Parameters:*

*item* The item to push to the start of the list

### E getFirst ()

Returns the first item of the list.

#### Pre condition

None

#### Post condition

None

#### *Returns:*

Returns first item of list, or null if empty

### E get (int *index*)

Returns item 'index'.

#### Pre condition

$0 < \text{index} < \text{list size}$

## Post condition

### ***Parameters:***

*index* Index of the item to return

### ***Returns:***

The 'index' item or null if out of boundry

---

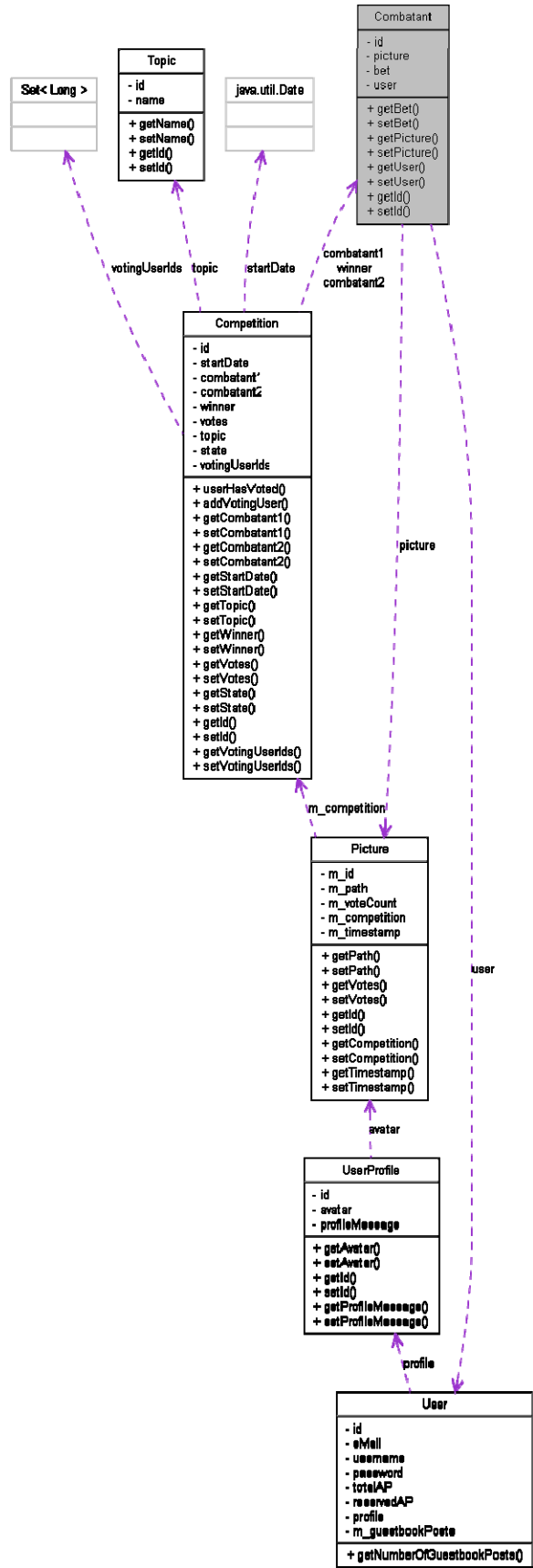
The documentation for this class was generated from the following file:

- [CircularList.java](#)

### ***Combatant Class Reference***

The [Combatant](#) class represents a user in a specific competition.

Collaboration diagram for Combatant:



## Public Member Functions

- int [getBet](#) ()
  - void [setBet](#) (int bet)
  - [Picture](#) [getPicture](#) ()
  - void [setPicture](#) ([Picture](#) picture)
  - [User](#) [getUser](#) ()
  - void [setUser](#) ([User](#) user)
  - Long [getId](#) ()
  - void [setId](#) (Long id)
- 

## Detailed Description

The [Combatant](#) class represents a user in a specific competition. This is an entity that can be saved in the database using hibernate. A combatant has a picture, a bet and user.

---

## Member Function Documentation

**int getBet ()**

**void setBet (int *bet*)**

**[Picture](#) getPicture ()**

**void setPicture ([Picture](#) *picture*)**

**[User](#) getUser ()**

**void setUser ([User](#) *user*)**

**Long getId ()**

**void setId (Long *id*)**

---

The documentation for this class was generated from the following file:

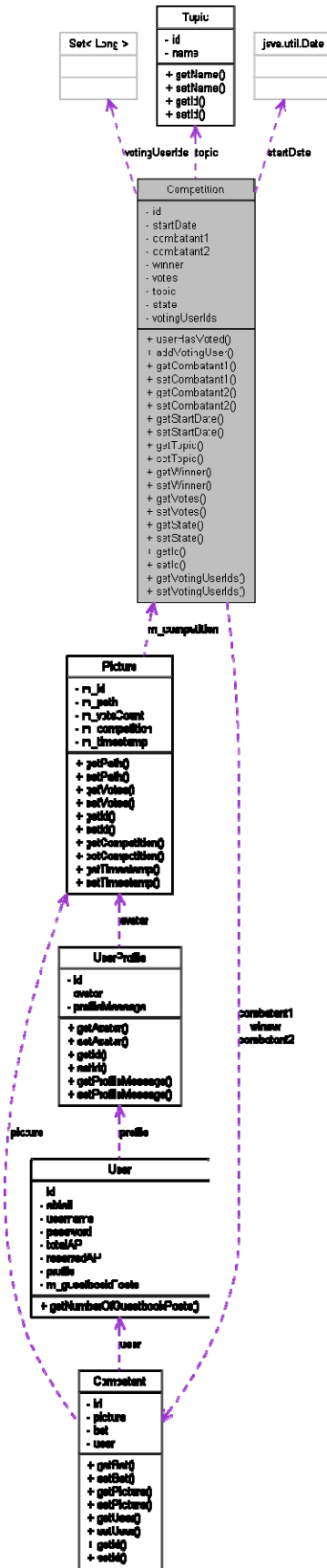
- [Combatant.java](#)



### ***Competition Class Reference***

A database entity that represents a competition.

Collaboration diagram for Competition:



## Public Member Functions

- boolean [userHasVoted](#) (Long userId)  
*Returns true if and only if the user with the specified id has voted for this competition.*
  - void [addVotingUser](#) ([User](#) user)  
*Adds a user to the list of voting users.*
  - [Combatant](#) [getCombatant1](#) ()
  - void [setCombatant1](#) ([Combatant](#) combatant1)
  - [Combatant](#) [getCombatant2](#) ()
  - void [setCombatant2](#) ([Combatant](#) combatant2)
  - Date [getStartDate](#) ()
  - void [setStartDate](#) (Date startDate)
  - [Topic](#) [getTopic](#) ()
  - void [setTopic](#) ([Topic](#) topic)
  - [Combatant](#) [getWinner](#) ()
  - void [setWinner](#) ([Combatant](#) winner)
  - int [getVotes](#) ()
  - void [setVotes](#) (int votes)
  - String [getState](#) ()
  - void [setState](#) (String state)
  - Long [getId](#) ()
  - void [setId](#) (Long id)
  - Set< Long > [getVotingUserIds](#) ()
  - void [setVotingUserIds](#) (Set< Long > votingUserIds)
- 

## Detailed Description

A database entity that represents a competition.  
This class will be saved in the database using hibernate.

---

## Member Function Documentation

### boolean [userHasVoted](#) (Long *userId*)

Returns true if and only if the user with the specified id has voted for this competition.

#### ***Parameters:***

*userId* The id of the user.

#### ***Returns:***

True if the user has voted in the competition.

**void addVotingUser ([User](#) *user*)**

Adds a user to the list of voting users.

***Parameters:***

*user* The user to add.

**[Combatant](#) getCombatant1 ()**

**void setCombatant1 ([Combatant](#) *combatant1*)**

**[Combatant](#) getCombatant2 ()**

**void setCombatant2 ([Combatant](#) *combatant2*)**

**Date getStartDate ()**

**void setStartDate (Date *startDate*)**

**[Topic](#) getTopic ()**

**void setTopic ([Topic](#) *topic*)**

**[Combatant](#) getWinner ()**

**void setWinner ([Combatant](#) *winner*)**

**int getVotes ()**

**void setVotes (int *votes*)**

**String getState ()**

**void setState (String *state*)**

**Long getId ()**

**void setId (Long *id*)**

**Set<Long> getVotingUserIds ()**

**void setVotingUserIds (Set< Long > *votingUserIds*)**

---

The documentation for this class was generated from the following file:

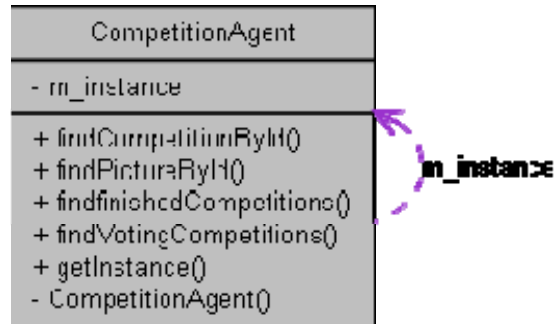
- [Competition.java](#)



## CompetitionAgent Class Reference

An [agent](#) with the responsibility to retrieve Competitions from the database.

Collaboration diagram for CompetitionAgent:



## Public Member Functions

- [Competition](#) [findCompetitionById](#) (long id)  
*Finds the competition object with the given id.*
- [Picture](#) [findPictureById](#) (long id)  
*Finds the picture with the given id.*
- List< [Competition](#) > [findfinishedCompetitions](#) (Date date, String Combatant1, String Combatant2, String winner, String looser, String topic, int votes)  
*Returns all finished competitions that matches the given parameters.*
- List< [Competition](#) > [findVotingCompetitions](#) (Date date, String Combatant1, String Combatant2, String topic, boolean goldenVote)  
*Returns all competitions in the voting phase that matches the given parameters.*

## Static Public Member Functions

- static [CompetitionAgent](#) [getInstance](#) ()  
*This method returns the singleton instance of the class.*

---

## Detailed Description

An [agent](#) with the responsibility to retrieve Competitions from the database.

The class is thread safe.

### **Author:**

Marcus Bergenlid Ali Mosavian

---

## Member Function Documentation

### [Competition](#) findCompetitionById (long *id*)

Finds the competition object with the given id.

#### Pre condition>

None

#### Post condition>

None

#### **Parameters:**

*id* The unique id of the competition to retrieve

#### **Returns:**

The competition with the specified id. null, if there are no competition with this id.

### [Picture](#) findPictureById (long *id*)

Finds the picture with the given id.

#### Pre condition>

None

#### Post condition>

None

#### **Parameters:**

*id* The unique id of the picture to retrieve

#### **Returns:**

The picture with the specified id. null, if there are no pictures with this id.

### List< [Competition](#) > findfinishedCompetitions (Date *date*, String *Combatant1*, String *Combatant2*, String *winner*, String *looser*, String *topic*, int *votes*)

Returns all finished competitions that matches the given parameters.

If one of the parameters is null the method will ignore that in the database query. The two combatant parameters can match one of the competitions combatant.

#### Pre condition>

None

#### Post condition>

None

#### **Parameters:**

*date* The startDate of the competitions

*Combatant1* The name of the first combatant.  
*Combatant2* The name of the second combatant.  
*winner* The name of the winner.  
*loser* The name of the loser.  
*topic* The topic

**Returns:**

A list of competitions matching the criteria above.

**List< [Competition](#) > findVotingCompetitions (Date *date*, String *Combatant1*, String *Combatant2*, String *topic*, boolean *goldenVote*)**

Returns all competitions in the voting phase that matches the given parameters.

If one of the parameters is null the method will ignore that in the database query. The two combatant parameters can match one of the competitions combatant.

**Pre condition>**

None

**Post condition>**

None

**Parameters:**

*date* The startDate of the competitions  
*Combatant1* The name of the first combatant.  
*Combatant2* The name of the second combatant.  
*topic* The topic  
*goldenVote* find golden vote competitions only.

**Returns:**

A list of competitions matching the criteria above.

**static [CompetitionAgent](#) getInstance () [static]**

This method returns the singleton instance of the class.

**Pre condition>**

Either singleton instance has been created or not

**Post condition>**

Singleton instance is created, otherwise nothing.

**Returns:**

The instance of competitionAgent.

---

The documentation for this class was generated from the following file:

- [CompetitionAgent.java](#)



## *GeneralEnum Class Reference*

A class for common enums.

### Public Types

- enum [COMPARATOR](#)
- 

### Detailed Description

A class for common enums.

#### *Author:*

Ali Mosavian

---

### Member Enumeration Documentation

#### enum [COMPARATOR](#)

Enumerator:

*LESS*  
*GREATER*  
*EQUAL*  
*LESS\_OR\_EQUAL*  
*GREATER\_OR\_EQUAL*

---

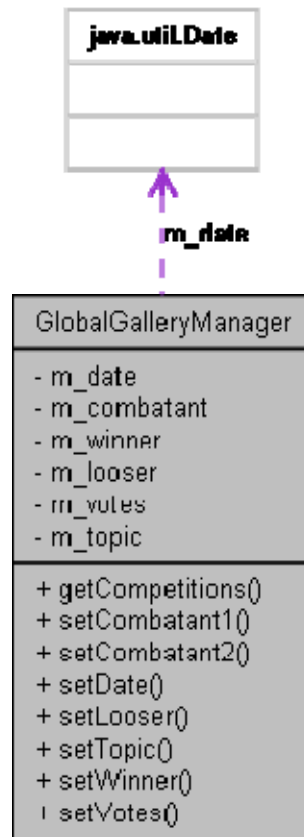
The documentation for this class was generated from the following file:

- [GeneralEnum.java](#)

## GlobalGalleryManager Class Reference

This class is responsible for processing every request that can be made on the global gallery page.

Collaboration diagram for GlobalGalleryManager:



## Public Member Functions

- List< [Competition](#) > [getCompetitions](#) ()  
*This method is called by GlobalGalleryJSP.*
- void [setCombatant1](#) (String combatant)  
*Sets the first battle participant property.*
- void [setCombatant2](#) (String combatant)  
*Sets the second battle participant property.*
- void [setDate](#) (Date date)  
*Sets the battle date property.*
- void [setLooser](#) (String looser)  
*Sets the looser property.*
- void [setTopic](#) (String topic)  
*Sets the topic property.*
- void [setWinner](#) (String winner)

*Sets the winner property.*

- void [setVotes](#) (int votes)  
*Sets the votes property.*

---

## Detailed Description

This class is responsible for processing every request that can be made on the global gallery page.

### ***Author:***

Marcus Bergenlid Ali Mosavian

---

## Member Function Documentation

### List<[Competition](#)> getCompetitions ()

This method is called by GlobalGalleryJSP.

The method will then use CompetitionAgent to find all Competitions that matches the fields that are set. The result will be in the field competitions, a list of competitions that the JSP-page can get later.

#### **Pre condition>**

None

#### **Post condition>**

None

#### ***Returns:***

Either list with competition or empty list

### void setCombatant1 (String *combatant*)

Sets the first battle participant property.

#### **Pre condition>**

None

#### **Post condition>**

The first battle participant is set.

#### ***Parameters:***

*combatant* Name of the first participant

### void setCombatant2 (String *combatant*)

Sets the second battle participant property.

**Pre condition>**

None

**Post condition>**

The second battle participant is set

***Parameters:***

*combatant* Name of the second participant

**void setDate (Date *date*)**

Sets the battle date property.

**Pre condition>**

None

**Post condition>**

The battle date is set

***Parameters:***

*date* A date

**void setLooser (String *looser*)**

Sets the looser property.

**Pre condition>**

None

**Post condition>**

The property is set.

***Parameters:***

*looser* Name of the looser

**void setTopic (String *topic*)**

Sets the topic property.

**Pre condition>**

None

**Post condition>**

The property is set.

***Parameters:***

*topic* The topic property to use

### **void setWinner (String *winner*)**

Sets the winner property.

#### **Pre condition>**

None

#### **Post condition>**

The property is set.

#### ***Parameters:***

*winner* The name of the winner

### **void setVotes (int *votes*)**

Sets the votes property.

#### **Pre condition>**

None

#### **Post condition>**

The property is set.

#### ***Parameters:***

*votes* Number of votes

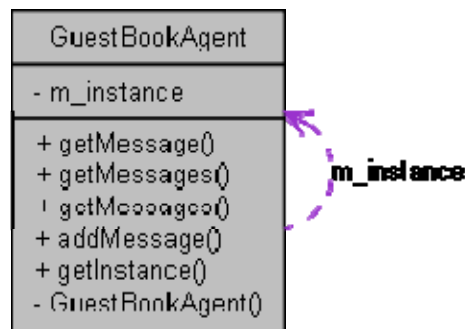
---

The documentation for this class was generated from the following file:

- [GlobalGalleryManager.java](#)

## GuestBookAgent Class Reference

Collaboration diagram for GuestBookAgent:



### Public Member Functions

- [GuestBookMessage](#) `getMessage` (long id)  
*Find and return the guestbook messages with the id id .*
- List< [GuestBookMessage](#) > `getMessages` (long userID, int count)  
*Find and return a list of guestbook messages starting from the latest time wise and the specified amount (maximum) of post backward time wise.*
- List< [GuestBookMessage](#) > `getMessages` (long userID, long localPostID, int count)  
*Find and return a list of guestbook messages with the first one being less then or equal to the specified local post id.*
- void `addMessage` ([GuestBookMessage](#) m)  
*Add a guestbook [message](#) to the database.*

### Static Public Member Functions

- static [GuestBookAgent](#) `getInstance` ()

---

### Member Function Documentation

#### [GuestBookMessage](#) `getMessage` (long id)

Find and return the guestbook messages with the id id .

#### Pre condition(s)

None

#### Post condition(s)

If there exists a [message](#) with the specified id, it is returned.

#### **Parameters:**

*id* The guesbook [message](#) id.

**Returns:**

The guestbook [message](#).

**List<[GuestBookMessage](#)> getMessages (long *userID*, int *count*)**

Find and return a list of guestbook messages starting from the latest time wise and the specified amount (maximum) of post backward time wise.

**Pre condition(s)**

**None Post condition(s)**

None

**Parameters:**

*userID* The user id of the guestbook owner

*count* The maximum number of posts to return

**Returns:**

A list of Guestbooks messages sorted descending after 'localID'. The first [message](#) will have the largest localID, meaning that it's the latest time wise. All the rest will have values less then it. If there are non null will be returned.

**List<[GuestBookMessage](#)> getMessages (long *userID*, long *localPostID*, int *count*)**

Find and return a list of guestbook messages with the first one being less then or equal to the specified local post id.

And going backward (time wise) with a maximum of 'count' posts.

**Pre condition(s)**

**None Post condition(s)**

None

**Parameters:**

*userID* The user id of the guestbook owner

*localPostID* The local ID of the first (last time wise) guestbook post.

*count* The maximum number of posts to return

**Returns:**

A list of Guestbooks messages sorted descending after local post ID. The first returned [message](#) will be exactly or less then 'localPostID' and the rest will be all be less.

**void addMessage ([GuestBookMessage](#) *m*)**

Add a guestbook [message](#) to the database.

## Pre condition(s)

## None Post condition(s)

None

### *Parameters:*

*m* The guest book [message](#) to add to the guest book

### *Returns:*

Nothing

**static [GuestBookAgent](#) getInstance () [static]**

## Description

Returns the [GuestBookAgent](#) instance. It makes sure that there is only one [UserAgent](#) instance, if there is none, one is created. Implements the singleton design pattern.

## Pre condition(s)

None

## Post condition(s)

The singleton instance of the [GustBookAgent](#) is returned.

### *Returns:*

The [GuestBookAgent](#) instance.

---

The documentation for this class was generated from the following file:

- [GuestBookAgent.java](#)



## *GuestBookManager Class Reference*

This class handles users guestbooks.

### Public Member Functions

- [Message setNewMessage \(\)](#)  
*The method creates a new post in a users guestbook Before use, set the properties senderID recieverID messageText.*
  - List< [GuestBookMessage](#) > [getMessages \(\)](#)  
*The method retrieves all post in the requested users guestbook, starting from the specified timestamp and max 'messageCount' messages forward.*
  - void [setGuestBookUserID](#) (int userID)
  - void [setSenderID](#) (int userID)
  - void [setReceiverID](#) (int userID)
  - void [setMessageText](#) (String text)
  - void [setMessageCount](#) (int count)
  - void [setMessageFirst](#) (long timestamp)
- 

### Detailed Description

This class handles users guestbooks.

Both posting and retrieving user messages. It's a java bean, so its properties has to be set before attempting to do anything.

#### ***Author:***

Per Almquist Ali Mosavian

---

### Member Function Documentation

#### **[Message setNewMessage \(\)](#)**

The method creates a new post in a users guestbook Before use, set the properties senderID recieverID messageText.

#### **Pre condition**

#### **Properties (senderID, recieverID, messageText ) are set Post condition**

A new post in the the guestbook of user 'senderID' from user 'receiverID' with the text 'messageText'

#### ***Returns:***

MessageOK on success MessageError on failing

## List<[GuestBookMessage](#)> getMessages ()

The method retrieves all post in the requested users guestbook, starting from the specified timestamp and max 'messageCount' messages forward.

### Pre condition

Properties (guestbookUserID, messageFirst, messageCount ) are set

### Post condition

None

### *Returns:*

List of GuestBookMessages

**void setGuestBookUserID (int *userID*)**

**void setSenderID (int *userID*)**

**void setReceiverID (int *userID*)**

**void setMessageText (String *text*)**

**void setMessageCount (int *count*)**

**void setMessageFirst (long *timestamp*)**

---

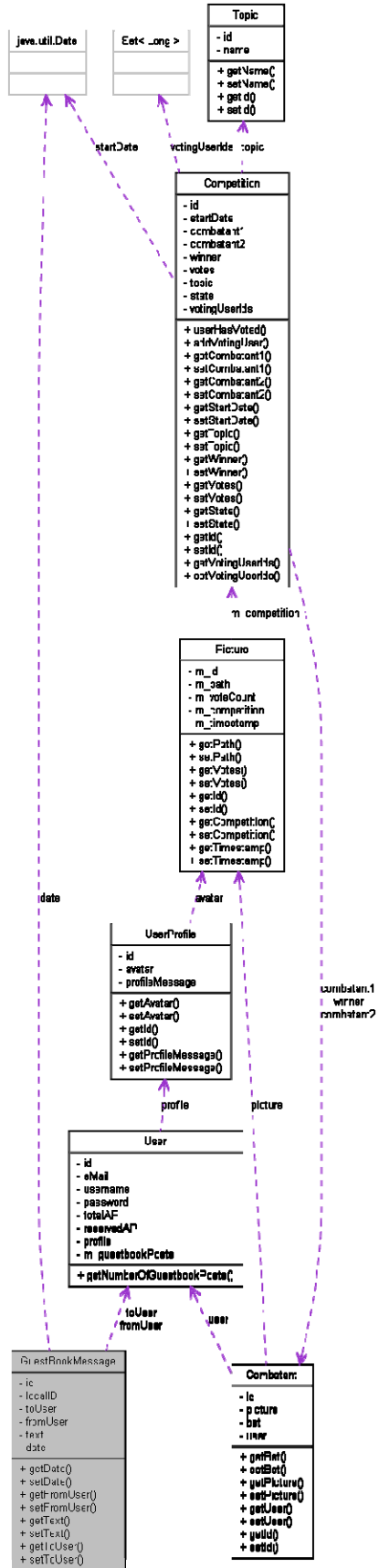
The documentation for this class was generated from the following file:

- [GuestBookManager.java](#)

### ***GuestBookMessage Class Reference***

A database entity that represents a guestbook post.

Collaboration diagram for GuestBookMessage:



## Public Member Functions

- Date [getDate](#) ()
  - void [setDate](#) (Date *date*)
  - [User](#) [getFromUser](#) ()
  - void [setFromUser](#) ([User](#) *fromUser*)
  - String [getText](#) ()
  - void [setText](#) (String *text*)
  - [User](#) [getToUser](#) ()
  - void [setToUser](#) ([User](#) *toUser*)
- 

## Detailed Description

A database entity that represents a guestbook post.  
This class will be saved in the database using hibernate.

---

## Member Function Documentation

**Date** [getDate](#) ()

**void** [setDate](#) (Date *date*)

[User](#) [getFromUser](#) ()

**void** [setFromUser](#) ([User](#) *fromUser*)

**String** [getText](#) ()

**void** [setText](#) (String *text*)

[User](#) [getToUser](#) ()

**void** [setToUser](#) ([User](#) *toUser*)

---

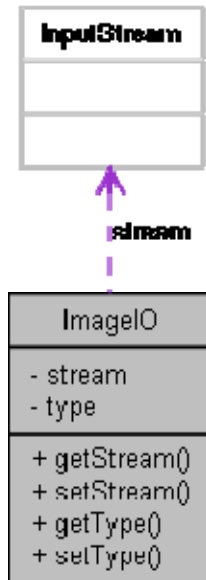
The documentation for this class was generated from the following file:

- [GuestBookMessage.java](#)

## ImageIO Class Reference

This class holds a stream to an [image](#) and is used for sending images to the client.

Collaboration diagram for ImageIO:



## Public Member Functions

- `InputStream` [getStream](#) ()
- `void` [setStream](#) (`InputStream` stream)
- `String` [getType](#) ()
- `void` [setType](#) (`String` type)

---

## Detailed Description

This class holds a stream to an [image](#) and is used for sending images to the client.

---

## Member Function Documentation

**InputStream** `getStream ()`

**void** `setStream (InputStream stream)`

**String** `getType ()`

**void** `setType (String type)`

---

The documentation for this class was generated from the following file:

- [ImageIO.java](#)

## *ImageManager Class Reference*

### Public Member Functions

- [ImageIO loadImage](#) (Long pictureId)  
*Finds the picture with the specified id from the database using CompetitionAgent.*
  - void [saveImage](#) ([ImageIO image](#), String path)  
*Saves the [image](#) given in the [ImageIO](#) at the specified path.*
- 

### Member Function Documentation

#### [ImageIO loadImage](#) (Long *pictureId*)

Finds the picture with the specified id from the database using CompetitionAgent.

It then finds out the path to the actual [image](#) data and creates an [ImageIO](#) object containing an InputStream to the [image](#) data and a type description of the [image](#).

#### **Parameters:**

*pictureId* The id of the picture.

#### **Returns:**

An object containing a stream to the [image](#) data.

#### void [saveImage](#) ([ImageIO image](#), String *path*)

Saves the [image](#) given in the [ImageIO](#) at the specified path.

#### **Parameters:**

*image* The [image](#) object to be saved.

*path* Where the [image](#) shall be saved.

---

The documentation for this class was generated from the following file:

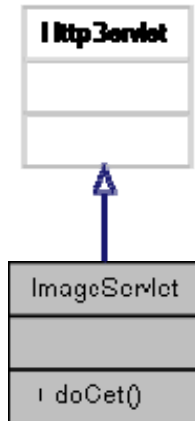
- [ImageManager.java](#)



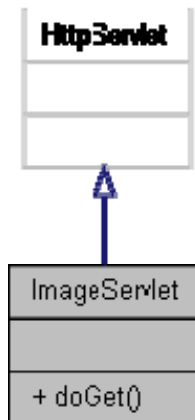
## ImageServlet Class Reference

This is the servlet class responsible for sending the binary [image](#) data to the client.

Inheritance diagram for ImageServlet:



Collaboration diagram for ImageServlet:



## Public Member Functions

- void [doGet](#) (HttpServletRequest request, HttpServletResponse response)  
*The request parameter will contain the id of the picture that shall be transferred to the client.*

---

## Detailed Description

This is the servlet class responsible for sending the binary [image](#) data to the client.

Every time the client need to download a picture this is the class it shall use.

---

## Member Function Documentation

**void doGet (HttpServletRequest *request*, HttpServletResponse *response*)**

The request parameter will contain the id of the picture that shall be tranfered to the client.

The method wil simply use [ImageManager](#) to retrieve a Stream to the binary data and then send it to the client by using the response parameter.

***Parameters:***

*request* The request

*response* The response

---

The documentation for this class was generated from the following file:

- [ImageServlet.java](#)

## ***ListHandler Class Reference***

This class stores [chat](#) messages in a circular queue and provides messages to retrieve them based on timestamp.

### **Public Member Functions**

- [ListHandler](#) (int size)  
*Initializes the list.*
  - void [add](#) ([MessageChat](#) m)  
*Adds a [message](#) to the circular [chat message](#) list, replacing the oldest [message](#).*
  - List< [MessageChat](#) > [getFrom](#) (long timestamp, int count)  
*The method retrieves all messages from timestamp *t* and *c* messages (if possible) forward.*
  - List< [MessageChat](#) > [getFromTo](#) (long first, long last)  
*The method retrieves all messages from timestamp *t* to timestamp *l*.*
- 

### **Detailed Description**

This class stores [chat](#) messages in a circular queue and provides messages to retrieve them based on timestamp.

This class is not thread safe.

#### ***Author:***

Ali Mosavian

---

### **Constructor & Destructor Documentation**

#### **[ListHandler](#) (int *size*)**

Initializes the list.

#### **Pre condition**

$0 < \text{size} < 1000$

#### **Post condition**

None

#### ***Parameters:***

*size* The maximum list size

---

### **Member Function Documentation**

## void add ([MessageChat](#) *m*)

Adds a [message](#) to the circular [chat message](#) list, replacing the oldest [message](#).

### Pre condition

None

### Post condition

First item of list is 'm', last item might have been replaced.

#### *Parameters:*

*m* The [message](#) to add

Referenced by ChatManager.addMessage().

Here is the caller graph for this function:



## List<[MessageChat](#)> getFrom (long *timestamp*, int *count*)

The method retrieves all messages from timestamp *t* and *c* messages (if possible) forward.

### Pre condition

$0 < \text{count} < \text{list size}$

### Post condition

None

#### *Parameters:*

*timestamp* Timestamp of the first [message](#)

*count* The number of messages to retrieve

#### *Returns:*

A list of ChatMessage, if none were found that fulfilled the requirements, an empty list is returned.

Referenced by ChatManager.getMessageFrom().

Here is the caller graph for this function:



## List<[MessageChat](#)> getFromTo (long *first*, long *last*)

The method retrieves all messages from timestamp *t* to timestamp *l*.

### Pre condition

None

### Post condition

None

#### *Parameters:*

*first* Timestamp of the first [message](#)

*last* Timestamp of the last [message](#)

**Returns:**

A list of ChatMessage, if none were found that fulfilled the requirements, an empty list is returned.  
Referenced by ChatManager.getMessageFromTo().

Here is the caller graph for this function:



---

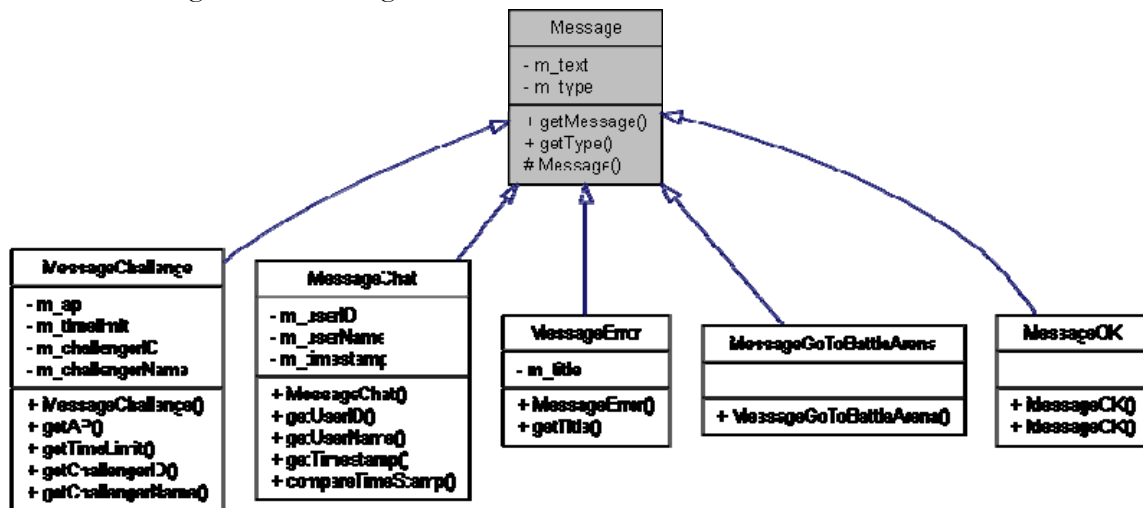
The documentation for this class was generated from the following file:

- [ListHandler.java](#)

## Message Class Reference

The base class for all types of messages.

Inheritance diagram for Message:



## Public Types

- enum [TYPE](#)

## Public Member Functions

- String [getMessage\(\)](#)  
*The method simply returns the text [message](#).*
- [TYPE](#) [getType\(\)](#)  
*Returns the [message](#) type.*

## Protected Member Functions

- [Message](#) (String text, [TYPE](#) t)  
*Constructor.*

---

## Detailed Description

The base class for all types of messages.

The class is thread safe

### **Author:**

Ali Mosavian

---

## Member Enumeration Documentation

enum [TYPE](#)

Enumerator:

*CHAT*  
*TEXT*  
*OK*  
*FAIL*  
*ERROR*  
*BEEN\_CHALLENGED*  
*CHALLENGE\_CANCELED*  
*ACCEPT*  
*GO\_TO\_BATTLE\_ARENA*  
*NEITHER*  
*CHALLENGER*  
*CHALLENGEE*

---

## Constructor & Destructor Documentation

[Message](#) (String *text*, [TYPE](#) *t*) [protected]

Constructor.

### Pre condition

None

### Post condition

Object initialized

### *Parameters:*

*text* The [message](#) text  
*t* [Message](#) type

---

## Member Function Documentation

### String getMessage ()

The methods simply returns the text [message](#).

### Pre condition

None

### Post condition

None

**Returns:**

The text [message](#) as String

**[TYPE](#) getType ()**

Returns the [message](#) type.

**Pre condition**

None

**Post condition**

None

**Returns:**

[Message](#) type

---

The documentation for this class was generated from the following file:

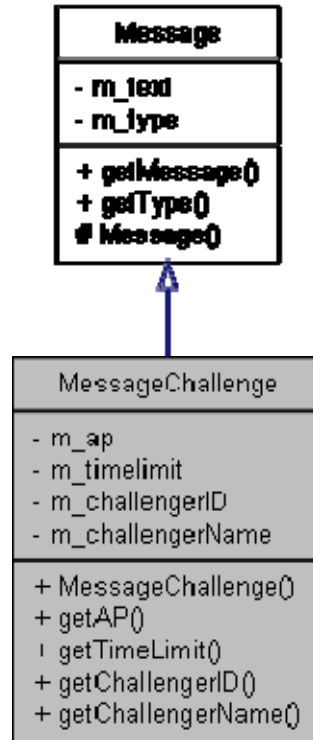
- [Message.java](#)



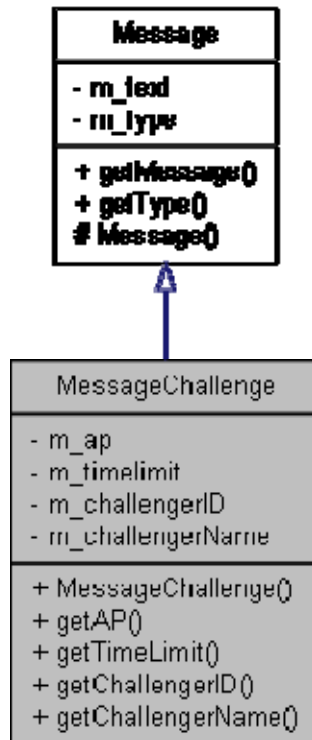
### *MessageChallenge Class Reference*

The class represents a challenge request [message](#) which is sent to the challengee in the event of being challenged.

Inheritance diagram for MessageChallenge:



Collaboration diagram for MessageChallenge:



## Public Member Functions

- [MessageChallenge](#) (int ap, int timelimit, long challengerID, String challengerName)  
*Constructor.*
- int [getAP](#) ()  
*Returns the bet AP in percent.*
- int [getTimeLimit](#) ()  
*Returns the battle timelimit in seconds.*
- long [getChallengerID](#) ()  
*Returns the challenger user ID.*
- String [getChallengerName](#) ()  
*Returns the challenger user name.*

## Detailed Description

The class represents a challenge request [message](#) which is sent to the challengee in the event of being challenged.

### **Author:**

Ali Mosavian

---

## Constructor & Destructor Documentation

**[MessageChallenge](#)** (int *ap*, int *timelimit*, long *challengerID*, String *challengerName*)

Constructor.

### Pre condition

None

### Post condition

Object initialized

#### ***Parameters:***

*ap* The betted AP in percent  
*timelimit* The battle timelimit in seconds  
*challengerID* The Challenger user ID  
*challengerName* The Challenger user name

---

## Member Function Documentation

### **int getAP ()**

Returns the bet AP in percent.

### Pre condition

None

### Post condition

None

#### ***Returns:***

AP

### **int getTimeLimit ()**

Returns the battle timelimit in seconds.

### Pre condition

None

### Post condition

None

#### ***Returns:***

Timelimit

### **long getChallengerID ()**

Returns the challenger user ID.

#### **Pre condition**

None

#### **Post condition**

None

#### ***Returns:***

User ID

### **String getChallengerName ()**

Returns the challenger user name.

#### **Pre condition**

None

#### **Post condition**

None

#### ***Returns:***

User name

---

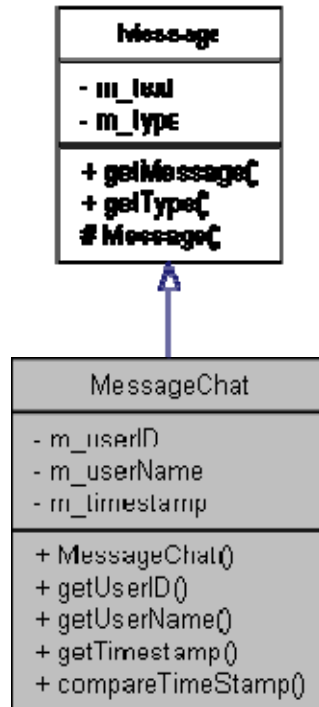
The documentation for this class was generated from the following file:

- [MessageChallenge.java](#)

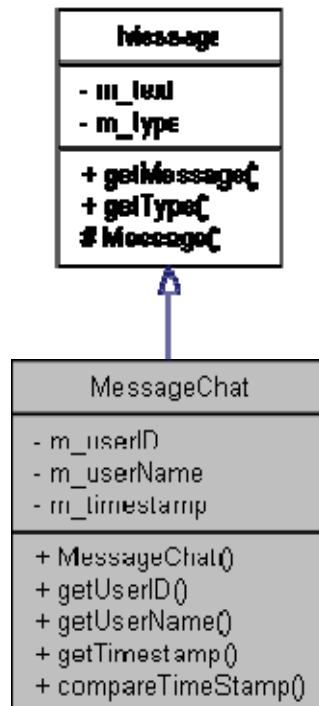
## MessageChat Class Reference

This class represents a chat [message](#).

Inheritance diagram for MessageChat:



Collaboration diagram for MessageChat:



## Public Member Functions

- [MessageChat](#) (int userID, String userName, String message)  
*Constructor, This method initializes the object, the timestamp property will be filled in automatically based on the number of seconds passed since midnight 1970 UTC.*
  - int [getUserID](#) ()  
*Returns the user id of the [message](#) author.*
  - String [getUserName](#) ()  
*Returns the (nick) name of the [message](#) author.*
  - long [getTimestamp](#) ()  
*Returns the [message](#) timestamp.*
  - long [compareTimeStamp](#) (long t)  
*This method will compare the timestamp t with the chat [message](#) timestamp.*
- 

## Detailed Description

This class represents a chat [message](#).

It holds the author user id, (nick) name and timestamp. The class is thread safe

### ***Author:***

Ali Mosavian

---

## Constructor & Destructor Documentation

### **[MessageChat](#) (int *userID*, String *userName*, String *message*)**

Constructor, This method initializes the object, the timestamp property will be filled in automatically based on the number of seconds passed since midnight 1970 UTC.

### **Pre condition**

None

### **Post condition**

Object initialized

### ***Parameters:***

*userID* The user ID of the [message](#) author

*userName* The (nick) name of the [message](#) author

*message* The actual [message](#)

---

## Member Function Documentation

### **int getUserID ()**

Returns the user id of the [message](#) author.

#### **Pre condition**

None

#### **Post condition**

None

#### **Returns:**

User id (integer)

### **String getUsername ()**

Returns the (nick) name of the [message](#) author.

pre cond: None

post cond: None

#### **Returns:**

Name (String)

### **long getTimestamp ()**

Returns the [message](#) timestamp.

#### **Pre condition**

None

#### **Post condition**

None

#### **Returns:**

Timestamp

### **long compareTimeStamp (long t)**

This method will compare the timestamp t with the chat [message](#) timestamp.

#### **Pre condition**

None

#### **Post condition**

None

#### **Parameters:**

t The timestamp to compare the [message](#)

**Returns:**

0, if t == message.timestamp  
-, if t < message.timestamp  
+, if t > message.timestamp

---

The documentation for this class was generated from the following file:

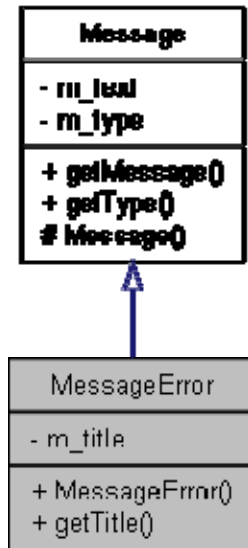
- [MessageChat.java](#)



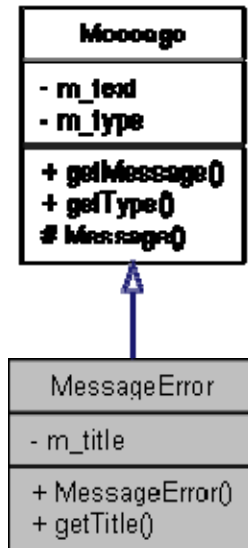
## MessageError Class Reference

This class represents an error [message](#) which is sent to the user.

Inheritance diagram for MessageError:



Collaboration diagram for MessageError:



## Public Member Functions

- [MessageError](#) (String title, String message)  
*Constructor.*
- String [getTitle](#) ()  
*Returns the error [message](#) title.*

---

## Detailed Description

This class represents an error [message](#) which is sent to the user.

### ***Author:***

Ali Mosavian

---

## Constructor & Destructor Documentation

### **MessageError** (String *title*, String *message*)

Constructor.

### **Pre condition**

None

### **Post condition**

Object initalized

### ***Parameters:***

*title*

[\*message\*](#)

---

## Member Function Documentation

### **String getTitle ()**

Returns the error [message](#) title.

### **Pre condition**

None

### **Post condition**

None

### ***Returns:***

String containing error title

---

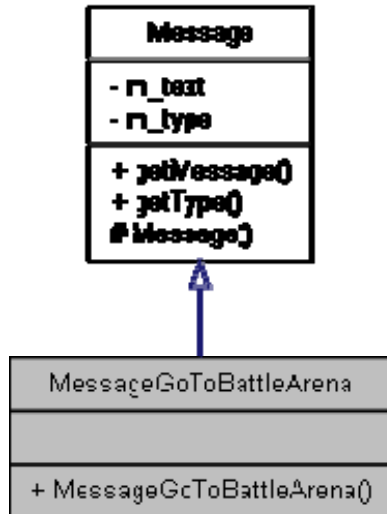
The documentation for this class was generated from the following file:

- [MessageError.java](#)

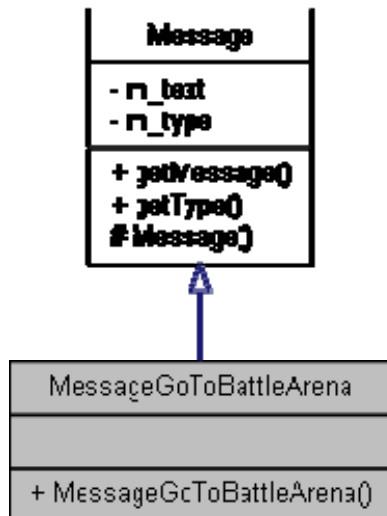
## *MessageGoToBattleArena Class Reference*

This class represent the [message](#) which tells the client to go the battle arena because they have an active battle.

Inheritance diagram for MessageGoToBattleArena:



Collaboration diagram for MessageGoToBattleArena:



## Public Member Functions

- [MessageGoToBattleArena \(\)](#)  
*Constructor.*

## Detailed Description

This class represent the [message](#) which tells the client to go the battle arena becuase they have an active battle.

### *Author:*

Ali Mosavian

---

## Constructor & Destructor Documentation

### [MessageGoToBattleArena \(\)](#)

Constructor.

### Pre condition

None

### Post condition

Object initialized

---

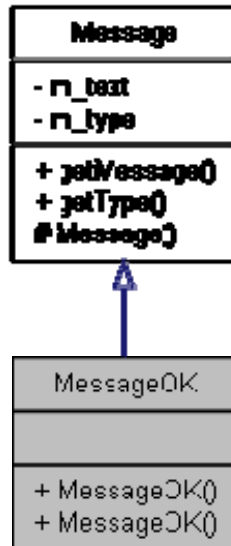
The documentation for this class was generated from the following file:

- [MessageGoToBattleArena.java](#)

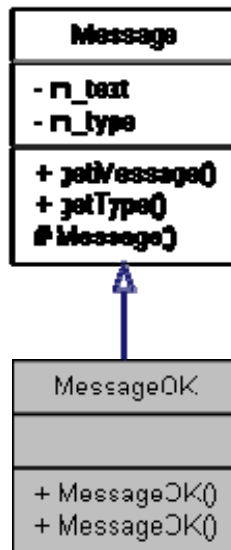
## MessageOK Class Reference

This class represent a success confirmation which the client will receive upon requesting an operation which succeeds.

Inheritance diagram for MessageOK:



Collaboration diagram for MessageOK:



## Public Member Functions

- [MessageOK \(\)](#)  
*Constructor.*
- [MessageOK \(String message\)](#)

Constructor, with a custom [message](#).

---

## Detailed Description

This class represent a success confirmation which the client will receive upon requesting an operation which succeeds.

### **Author:**

Ali Mosavian

---

## Constructor & Destructor Documentation

### **[MessageOK](#) ()**

Constructor.

### **Pre condition**

None

### **Post condition**

None

### **[MessageOK](#) (String *message*)**

Constructor, with a custom [message](#).

### **Pre condition**

None

### **Post condition**

None

### **Parameters:**

[message](#) The [message](#) to send to user.

---

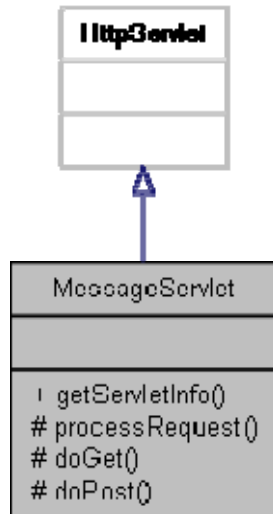
The documentation for this class was generated from the following file:

- [MessageOK.java](#)

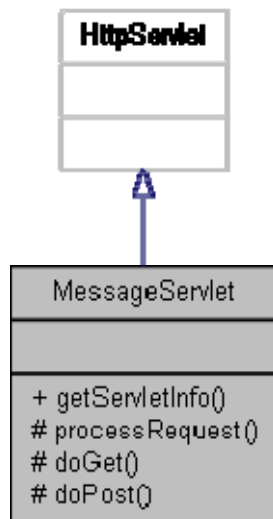
## *MessageServlet Class Reference*

All messages pass through this servlet.

Inheritance diagram for MessageServlet:



Collaboration diagram for MessageServlet:



## Public Member Functions

- String [getServletInfo\(\)](#)  
*Returns a short description of the servlet.*

## Protected Member Functions

- void [processRequest](#) (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException  
*Processes requests for both HTTP GET and POST methods.*
  - void [doGet](#) (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException  
*Handles the HTTP GET method.*
  - void [doPost](#) (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException  
*Handles the HTTP POST method.*
- 

## Detailed Description

All messages pass through this servlet.

### *Author:*

Ali Mosavian

---

## Member Function Documentation

**void processRequest (HttpServletRequest *request*, HttpServletResponse *response*) throws ServletException, IOException [protected]**

Processes requests for both HTTP GET and POST methods.

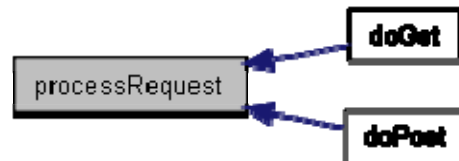
### *Parameters:*

*request* servlet request

*response* servlet response

Referenced by MessageServlet.doGet(), and MessageServlet.doPost().

Here is the caller graph for this function:



**void doGet (HttpServletRequest *request*, HttpServletResponse *response*) throws ServletException, IOException [protected]**

Handles the HTTP GET method.

### *Parameters:*

*request* servlet request



*response* servlet response

References `MessageServlet.processRequest()`.

Here is the call graph for this function:



**void doPost (HttpServletRequest *request*, HttpServletResponse *response*) throws ServletException, IOException [protected]**

Handles the HTTP POST method.

***Parameters:***

*request* servlet request

*response* servlet response

References `MessageServlet.processRequest()`.

Here is the call graph for this function:



**String getServletInfo ()**

Returns a short description of the servlet.

---

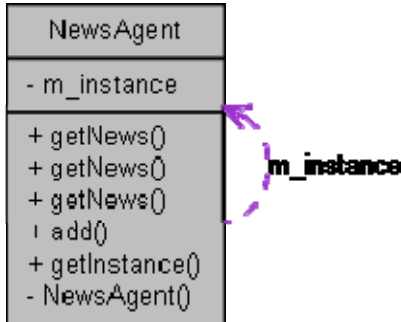
The documentation for this class was generated from the following file:

- [MessageServlet.java](#)

## NewsAgent Class Reference

The news [agent](#) is responsible for the retrieving of news messages and storing of new news messages in the database.

Collaboration diagram for NewsAgent:



## Public Member Functions

- [NewsMessage getNews](#) (long id)  
*Finds and retrieves a news [message](#) from the database.*
- List< [NewsMessage](#) > [getNews](#) (int count)  
*Finds and retrieves 'count' news messages from the database where the first will be the latest time wise and the rest will be in a time descending order.*
- List< [NewsMessage](#) > [getNews](#) (Date first, Date last)  
*Finds and retrieves a list of all messages created in a specific time period, and returned in a time descending order.*
- void [add](#) ([NewsMessage](#) m)  
*Add a new news [message](#) to the database.*

## Static Public Member Functions

- static [NewsAgent getInstance](#) ()  
*Returns the [NewsAgent](#) instance.*

---

## Detailed Description

The news [agent](#) is responsible for the retrieving of news messages and storing of new news messages in the database.

The class is thread safe.

### **Author:**

Per Almquist Peter Andersson Ali Mosavian

---

## Member Function Documentation

### [NewsMessage](#) getNews (long *id*)

Finds and retrieves a news [message](#) from the database.  
based on id.

#### Pre condition(s)

None

#### Post condition(s)

None

#### *Parameters:*

*id* The id of the news [message](#) to get.

#### *Returns:*

A news post with specified id or null if it doesn't exist.

### List<[NewsMessage](#)> getNews (int *count*)

Finds and retrieves 'count' news messages from the database where the first will be the latest time wise and the rest will be in a time descending order.

#### Pre condition(s)

None

#### Post condition(s)

None

#### *Parameters:*

*count* The maximum number of messages/post the retrieve.

#### *Returns:*

A list of news posts in a time descending order or null if there were none.

### List<[NewsMessage](#)> getNews (Date *first*, Date *last*)

Finds and retrieves a list of all messages created in a specific time period, and returned in a time descending order.

#### Pre condition(s)

None

#### Post condition(s)

None

#### *Parameters:*

*first* The date of the first news post.

*last* The date of the last news post.

**Returns:**

List of news messages in time descending order or null if none were found.

**void add ([NewsMessage](#) *m*)**

Add a new news [message](#) to the database.

**Pre condition(s)**

None

**Post condition(s)**

News [message](#) has been added to the database with time stamp set to the current time.

**Parameters:**

*m* The [message](#) to be added.

**static [NewsAgent](#) getInstance () [static]**

Returns the [NewsAgent](#) instance.

It makes sure that there is only one [UserAgent](#) instance, if there is none, one is created.  
Implements the singleton design pattern.

**Pre condition(s)**

None

**Post condition(s)**

The singleton instance of the [GustBookAgent](#) is returned.

**Returns:**

The [NewsAgent](#) instance.

---

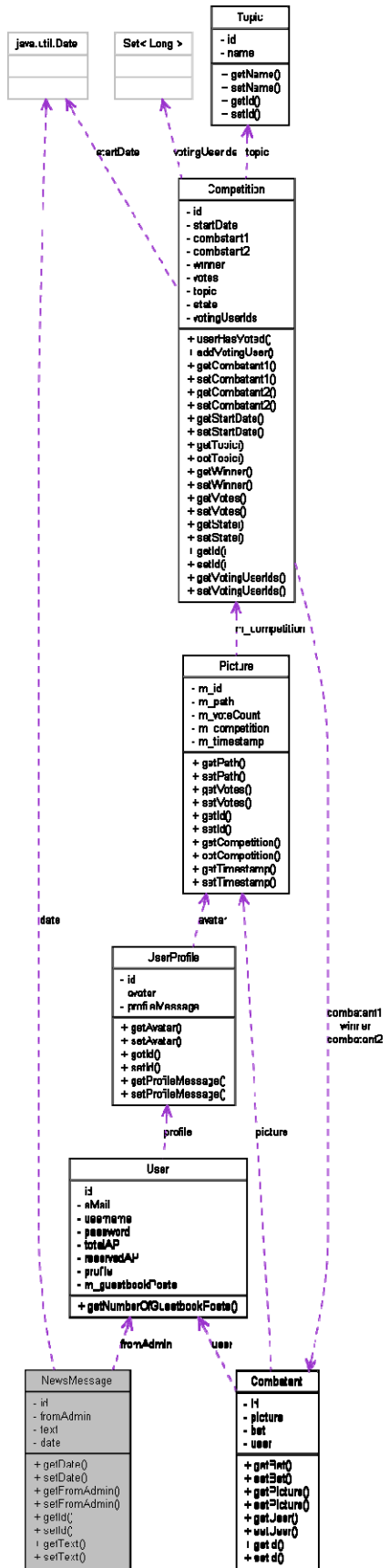
The documentation for this class was generated from the following file:

- [NewsAgent.java](#)

### ***NewsMessage Class Reference***

A database entity that represents a news [message](#).

Collaboration diagram for NewsMessage:



## Public Member Functions

- Date [getDate](#) ()  
*Returns the timestamp when the [message](#) was created.*
  - void [setDate](#) (Date date)  
*Set the time when the [message](#) was created.*
  - User [getFromAdmin](#) ()  
*Returns the admin that created the [message](#).*
  - void [setFromAdmin](#) (User fromAdmin)  
*Set the admin that created the [message](#).*
  - Long [getId](#) ()  
*Returns the id of the [message](#).*
  - void [setId](#) (Long id)  
*Set the id of the [message](#).*
  - String [getText](#) ()  
*Returns the contents of the news [message](#).*
  - void [setText](#) (String text)  
*Set the contents of the news [message](#).*
- 

## Detailed Description

A database entity that represents a news [message](#).  
This class will be saved in the database using hibernate.  
Javadoc by Pelle and Peter.

***Author:***

---

## Member Function Documentation

### Date [getDate](#) ()

Returns the timestamp when the [message](#) was created.

### void [setDate](#) (Date *date*)

Set the time when the [message](#) was created.

***Date:***

The time when the [message](#) was created.

### **User getFromAdmin ()**

Returns the admin that created the [message](#).

### **void setFromAdmin (User fromAdmin)**

Set the admin that created the [message](#).

The admin that created the [message](#).

### **Long getId ()**

Returns the id of the [message](#).

### **void setId (Long id)**

Set the id of the [message](#).

The id of the [message](#)

### **String getText ()**

Returns the contents of the news [message](#).

### **void setText (String text)**

Set the contents of the news [message](#).

The contents of the [message](#).

---

The documentation for this class was generated from the following file:

- [NewsMessage.java](#)



## *PersonalGalleryManager Class Reference*

### Public Member Functions

- List< [Picture](#) > [getPictures](#) ()
  - void [setUserId](#) (int *userId*)
  - void [setMessageCount](#) (int *c*)
  - void [setfirstTimestamp](#) (long *first*)
- 

### Member Function Documentation

List<[Picture](#)> [getPictures](#) ()

void [setUserId](#) (int *userId*)

void [setMessageCount](#) (int *c*)

void [setfirstTimestamp](#) (long *first*)

---

The documentation for this class was generated from the following file:

- [profile/PersonalGalleryManager.java](#)

## *PersonalGalleryManager Class Reference*

### Public Member Functions

- void [findPersonalGallery](#) ()  
*This method uses CompetitionAgent to find all Competitions that the user with id 'userId' has participated in.*
  - List< [Competition](#) > [getCompetitions](#) ()
  - void [setCompetitions](#) (List< [Competition](#) > competitions)
  - Long [getUserId](#) ()
  - void [setUserId](#) (Long userId)
- 

### Member Function Documentation

#### **void findPersonalGallery ()**

This method uses CompetitionAgent to find all Competitions that the user with id 'userId' has participated in.

The result will be put in the field 'competitions' and it will be sorted by date.

#### **List<[Competition](#)> getCompetitions ()**

#### **void setCompetitions (List< [Competition](#) > *competitions*)**

#### **Long getUserId ()**

#### **void setUserId (Long *userId*)**

---

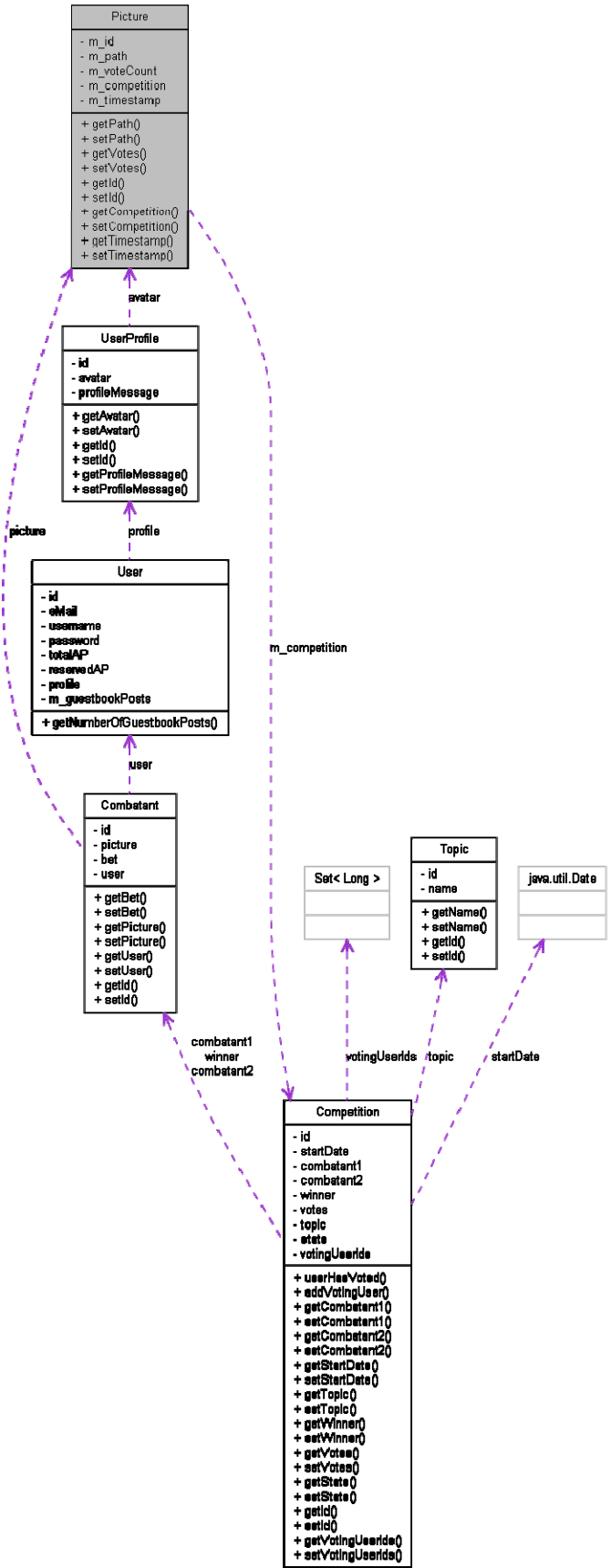
The documentation for this class was generated from the following file:

- [personalgallery/PersonalGalleryManager.java](#)

### ***Picture Class Reference***

Represents a picture that is stored in the database.

Collaboration diagram for Picture:



## Public Member Functions

- String [getPath](#) ()
  - void [setPath](#) (String path)
  - int [getVotes](#) ()
  - void [setVotes](#) (int votes)
  - Long [getId](#) ()
  - void [setId](#) (Long id)
  - [Competition](#) [getCompetition](#) ()
  - void [setCompetition](#) ([Competition](#) competition)
  - long [getTimestamp](#) ()
  - void [setTimestamp](#) (long t)
- 

## Detailed Description

Represents a picture that is stored in the database.

---

## Member Function Documentation

**String [getPath](#) ()**

**void [setPath](#) (String *path*)**

**int [getVotes](#) ()**

**void [setVotes](#) (int *votes*)**

**Long [getId](#) ()**

**void [setId](#) (Long *id*)**

**[Competition](#) [getCompetition](#) ()**

**void [setCompetition](#) ([Competition](#) *competition*)**

**long [getTimestamp](#) ()**

**void [setTimestamp](#) (long *t*)**

---

The documentation for this class was generated from the following file:

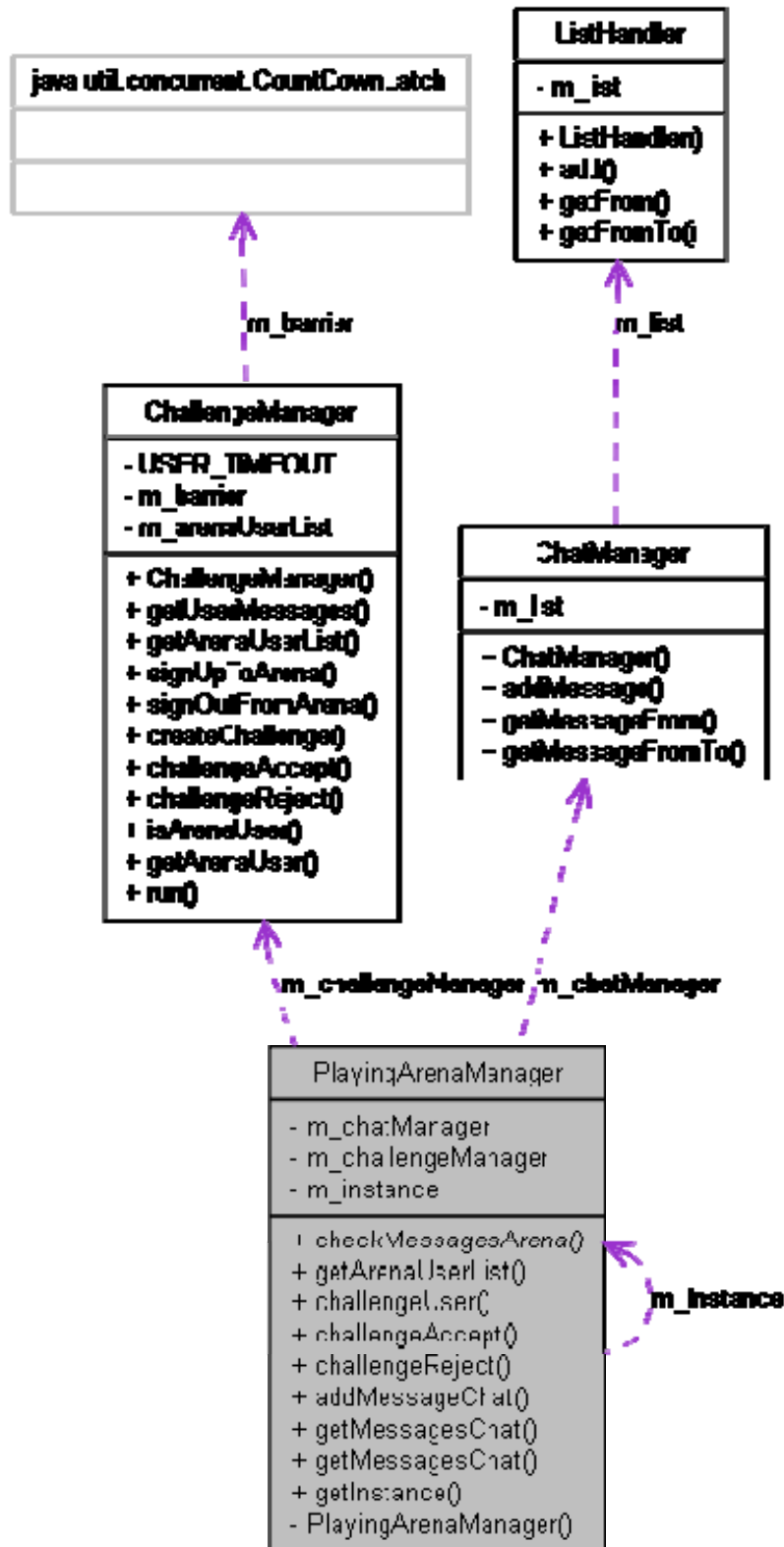
- [Picture.java](#)



### ***PlayingArenaManager Class Reference***

This class acts as a controller for the playing arena.

Collaboration diagram for PlayingArenaManager:





## Public Member Functions

- List< [Message](#) > [checkMessagesArena](#) (int userID)  
*Checks the users messages and updates the user so he/she doesn't time out.*
- Map< String, Long > [getArenaUserList](#) ()  
*Gets the signed in playing arena users as a sorted Map.*
- [Message challengeUser](#) (int challengerID, int challengeeID, int ap, int timelimit)  
*A [challenge](#) request.*
- [Message challengeAccept](#) (int userID)  
*Is called by [challenge](#) to accept a [challenge](#) request.*
- [Message challengeReject](#) (int userID)  
*Is called by a challenger/challenge to either reject or cancel are [challenge](#) request.*
- [Message addMessageChat](#) (int userID, String userName, String message)  
*Adds a [message](#) to [chat](#).*
- List< [MessageChat](#) > [getMessagesChat](#) (long first, int count)  
*Gets [chat](#) messages from a UTC timestamp and forward with maximum of 'count' messages.*
- List< [MessageChat](#) > [getMessagesChat](#) (long first, long last)  
*Get [chat](#) messages from a UTC timestamp to another UTC timestamp.*

## Static Public Member Functions

- static [PlayingArenaManager getInstance](#) ()  
*Will return a singleton instance of the class.*

---

## Detailed Description

This class acts as a controller for the playing arena.

It will handle all challenges and [chat](#) messages and makes sure no user does something they're not allowed. The class is thread safe.

### **Author:**

Ali Mosavian

---

## Member Function Documentation

### List<[Message](#)> [checkMessagesArena](#) (int *userID*)

Checks the users messages and updates the user so he/she doesn't time out.

### Pre condition

User is signed up in the arena.

## Post condition

None

### **Parameters:**

*userID* The id of the user

### **Returns:**

A list of messages, or null if user isn't signed up  
References ChallengeManager.isArenaUser().

Here is the call graph for this function:



## Map<String, Long> getArenaUserList ()

Gets the signed in playing arena users as a sorted Map.

## Pre condition

None

## Post condition

None

### **Returns:**

A sorted map with the arena users such as <User name, User ID>  
References ChallengeManager.getArenaUserList().

Here is the call graph for this function:



## [Message](#) challengeUser (int *challengerID*, int *challengeeID*, int *ap*, int *timelimit*)

A [challenge](#) request.

## Pre condition

- Challenger and [challenge](#) are signed up
- Neither have active [challenge](#) requests

## Post condition

- Challenge request is created
- A [message](#) is left for the challengee

### **Parameters:**

*challengerID* The user id of the challenger  
*challengeeID* The user id of the challengee

*ap* AP in percent  
*timelimit* The battle timelimit

**Returns:**

MessageOK or MessageError

**Message challengeAccept (int *userID*)**

Is called by [challenge](#) to accept a [challenge](#) request.

**Pre condition**

None

**Post condition**

None

**Parameters:**

*userID* The user id of the accepting user

**Returns:**

MessageGoToBattleArena or MessageError

**Message challengeReject (int *userID*)**

Is called by a challenger/challenge to either reject or cancel are [challenge](#) request.

**Pre condition**

User has challenged/been challenged

**Post condition**

None

**Parameters:**

*userID* The user id

**Returns:**

MessageOK or MessageError

**Message addMessageChat (int *userID*, String *userName*, String *message*)**

Adds a [message](#) to [chat](#).

The user needs to be signed in to the playing arena for this.

**Pre condition**

User is signed up in the arena

**Post condition**

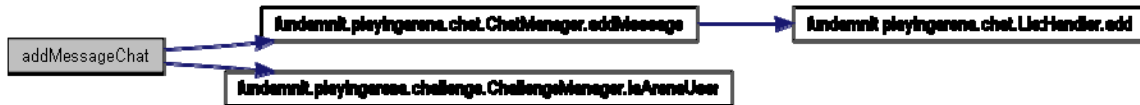
Message is among [chat](#) messages

### Parameters:

*userID* The user id  
*userName* The user name  
*message* The actual text [message](#)

### Returns:

MessageOK or MessageError  
References ChatManager.sendMessage(), and ChallengeManager.isArenaUser().  
Here is the call graph for this function:



### List<[MessageChat](#)> getMessagesChat (long *first*, int *count*)

Gets [chat](#) messages from a UTC timestamp and forward with maximum of 'count' messages.

### Pre condition

None

### Post condition

None

### Parameters:

*first* Timestamp of the first [message](#)  
*count* Maximum number of messages to fetch

### Returns:

A list of [chat](#) messages  
References ChatManager.getMessageFrom().  
Here is the call graph for this function:



### List<[MessageChat](#)> getMessagesChat (long *first*, long *last*)

Get [chat](#) messages from a UTC timestamp to another UTC timestamp.

### Pre condition

None

### Post condition

None

### Parameters:

*first* UTC timestamp of the first [message](#)  
*last* UTC timestamp of the last [message](#)

### Returns:

References ChatManager.getMessageFromTo().

Here is the call graph for this function:



### static [PlayingArenaManager](#) getInstance () [static]

Will return a singleton instance of the class.

Method is thread safe

### Pre condition

None

### Post condition

Instance is created if it hadn't previously been.

### *Returns:*

Instance of the class

---

The documentation for this class was generated from the following file:

- [PlayingArenaManager.java](#)

## *ProfileManager Class Reference*

### Public Member Functions

- void [findUserInformation](#) ()
  - void [findPersonalGallery](#) ()  
*Fetch the user's personal gallery.*
  - void [findGuestBook](#) ()  
*Fetch the user's guestbook.*
- 

### Detailed Description

#### Description

Find the necessary data needed in order to generate a user [profile](#) page. This data concerns a specific user's general information, guestbook and personal gallery.

---

### Member Function Documentation

#### void [findUserInformation](#) ()

##### Description

Fetch the user's general information. That is the user's

- username
- [profile message](#)
- avatar picture
- total AP
- reserved AP
- number of participated competitions

##### Pre

##### None. Post

None.

#### void [findPersonalGallery](#) ()

Fetch the user's personal gallery.

Uses [PersonalGalleryManager](#)

#### void [findGuestBook](#) ()

Fetch the user's guestbook.

Uses [GuestBookManager](#)

---

The documentation for this class was generated from the following file:

- [ProfileManager.java](#)

## *ProfilePictureManager Class Reference*

### Public Member Functions

- void [postPicture](#) (Long *userId*, InputStream *picture*)
- 

### Detailed Description

#### Description

This class retrieves pictures from [ProfilePictureServlet](#) and binds them to the correct [User](#)'s [UserProfile](#) avatar and saves them in the file system.

---

### Member Function Documentation

#### void postPicture (Long *userId*, InputStream *picture*)

#### Description

This method is invoked by [ProfilePictureServlet](#), the *userId* is the id of the user's. The method will create a [Picture](#) and bind it to the correct [UserProfile](#) by using [UserAgent](#).

Then the picture will be saved in the filesystem with the help of [ImageIO](#).

#### Pre

**The user has uploaded a picture to be used as his/her avatar picture.**

#### Post

The avatar picture is added to the user's [profile](#).

#### *Parameters:*

*userId* The user's id in the system.

*picture* The avatar picture.

---

The documentation for this class was generated from the following file:

- [ProfilePictureManager.java](#)



## ***ProfilePictureServlet Class Reference***

This class enables the user to upload his/her avatar picture.

### **Public Member Functions**

- void [doPost](#) (HttpServletRequest req, HttpServletResponse resp)  
*The request parameter will contain binary data representing the picture sent by the user's web browser.*
- 

### **Detailed Description**

This class enables the user to upload his/her avatar picture.

---

### **Member Function Documentation**

#### **void doPost (HttpServletRequest *req*, HttpServletResponse *resp*)**

The request parameter will contain binary data representing the picture sent by the user's web browser.

The method will obtain an InputStream to the binary data from the request parameter and the user's userId from the session.

The [ProfilePictureManager](#) is then responsible for saving the picture in the file system.

#### ***Parameters:***

*req* The Request parameter.

*resp* The Response parameter.

---

The documentation for this class was generated from the following file:

- [ProfilePictureServlet.java](#)

## *RegisterManager Class Reference*

### Public Member Functions

- void [addUser](#) ()
  - String [getEMail](#) ()  
*Return the e-mail address.*
  - void [setEMail](#) (String mail)  
*Set the e-mail address.*
  - String [getPassword](#) ()  
*Return the password.*
  - void [setPassword](#) (String password)  
*Set the password.*
  - int [getTotalAP](#) ()  
*Return the total AP.*
  - void [setTotalAP](#) (int totalAP)  
*Set the total AP.*
  - String [getUsername](#) ()  
*Return the username.*
  - void [setUsername](#) (String username)  
*Set the username.*
- 

### Detailed Description

#### Description

A [register](#) manager handles the registration process of new users to the system.

---

### Member Function Documentation

#### void addUser ()

#### Description

Create and add a new user to a the system. Uses the JavaBean convention to store and retrieve information about the user. A new user is created by instantiating a [RegisterManager](#) object and it's member fields

- eMail
- username
- password
- totalAP

then calling this method.

## Pre

### The member fields need to be filled. Post

If all member fields are filled by appropriate values (see [User](#) to see what is an appropriate value for each field). Then a new [User](#) is created.

### String getEmail ()

Return the e-mail address.

#### **Returns:**

The e-mail address.

### void setEmail (String mail)

Set the e-mail address.

#### **Parameters:**

*mail* The E-mail address.

### String getPassword ()

Return the password.

#### **Returns:**

The password.

### void setPassword (String password)

Set the password.

#### **Parameters:**

*password* The password.

### int getTotalAP ()

Return the total AP.

#### **Returns:**

The total AP.

### void setTotalAP (int totalAP)

Set the total AP.

***Parameters:***

*totalAP* The total AP.

**String getUsername ()**

Return the username.

***Returns:***

The username.

**void setUsername (String *username*)**

Set the username.

***Parameters:***

*username* The username.

---

The documentation for this class was generated from the following file:

- [RegisterManager.java](#)

## *SearchManager Class Reference*

### Public Member Functions

- void [findUsers](#) ()
- 

### Detailed Description

#### Description

A [search](#) manager handles [search](#) requests from the SearchJSP.

---

### Member Function Documentation

#### void findUsers ()

#### Description

Find users that meets the criteria made up by the fields

- Name
- Number of competitions the user has participated in
- Users total AP
- Number of competitions the user has won

the comparators

- '>' (greater than)
- '>=' (greater than or equal to)
- '<=' (less than or equal to)
- '<' (less than)

#### Pre

#### None. Post

The list users is updated.

---

The documentation for this class was generated from the following file:

- [SearchManager.java](#)

## *Topic Class Reference*

Represents a topic that is used in the competitions.

### Public Member Functions

- String [getName](#) ()
  - void [setName](#) (String name)
  - Long [getId](#) ()
  - void [setId](#) (Long id)
- 

### Detailed Description

Represents a topic that is used in the competitions.

---

### Member Function Documentation

**String [getName](#) ()**

**void [setName](#) (String *name*)**

**Long [getId](#) ()**

**void [setId](#) (Long *id*)**

---

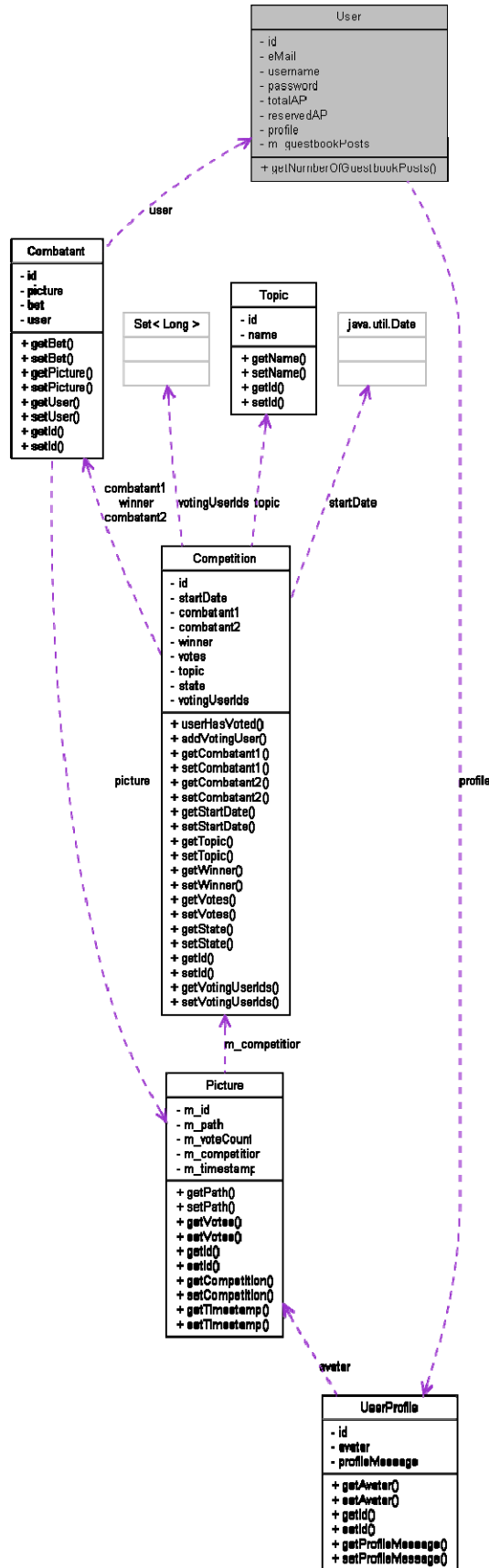
The documentation for this class was generated from the following file:

- [Topic.java](#)

### ***User Class Reference***

A database entity that represents a user.

Collaboration diagram for User:





## Public Member Functions

- `int getNumberOfGuestbookPosts ()`  
*Returns the number of guestbook posts for this user.*
- 

## Detailed Description

A database entity that represents a user.

This class will be saved in the database using hibernate.

---

## Member Function Documentation

### `int getNumberOfGuestbookPosts ()`

Returns the number of guestbook posts for this user.

Everytime a new guestbook [message](#) is added this method will be called. The purpose is to keep track of the ID's of the messages local to this user. This is to ensure that messages will be stored and retrieved in chronological order when the guestbook is fetched from the database.

---

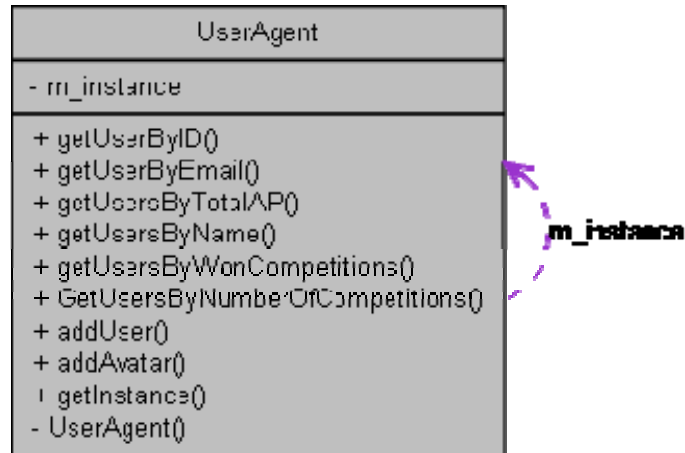
The documentation for this class was generated from the following file:

- [User.java](#)

## UserAgent Class Reference

The user [agent](#) is responsible for the retrieving and storing information about a user in the database.

Collaboration diagram for UserAgent:



## Public Member Functions

- [User](#) [getUserByID](#) (long userID)  
*Finds the user with a specific user id.*
- [User](#) [getUserByEmail](#) (String email)  
*Finds the user with a specific user email.*
- List< [User](#) > [getUsersByTotalAP](#) ([GeneralEnum.COMPARATOR](#) c, int value)  
*Find users with a total AP that meets the criteria made up by the comparator c and the value v of.*
- List< [User](#) > [getUsersByName](#) (String n)  
*Finds all user that match the given name.*
- List< [User](#) > [getUsersByWonCompetitions](#) ([GeneralEnum.COMPARATOR](#) c, int value)  
*Find users with the number of won competitions that meets the criteria made up by the comparator c and the value v of.*
- List< [User](#) > [GetUsersByNumberOfCompetitions](#) ([GeneralEnum.COMPARATOR](#) c, int value)  
*Find users with the number of participated competitions that meets the criteria made up by the comparator c and the value v.*
- void [addUser](#) ([User](#) u)  
*Add a new user to the database.*
- void [addAvatar](#) (long userID, [Picture](#) p)  
*Add an avatar picture to the user's [profile](#).*

## Static Public Member Functions

- static [UserAgent](#) [getInstance](#) ()  
*Returns the [UserAgent](#) instance.*

---

## Detailed Description

The user [agent](#) is responsible for the retrieving and storing information about a user in the database. The class is thread safe.

### ***Author:***

Per Almquist Ali Mosavian

---

## Member Function Documentation

### **User [getUserByID](#) (long *userID*)**

Finds the user with a specific user id.

### **Pre condition(s)**

User exists

### **Post condition(s)**

None

### ***Parameters:***

*userID* The user id to [search](#) for

### ***Returns:***

A user object or null if the user could not be found

### **User [getUserByEmail](#) (String *email*)**

Finds the user with a specific user email.

### **Pre condition(s)**

User exists

### **Post condition(s)**

None

### ***Parameters:***

*email* The email to [search](#) for

### ***Returns:***

A user object or null if the user could not be found

**List<[User](#)> getUsersByTotalAP ([GeneralEnum.COMPARATOR](#) *c*, int *value*)**

Find users with a total AP that meets the criteria made up by the comparator *c* and the value *v* of.

**Pre condition(s)**

None

**Post condition(s)**

None

***Parameters:***

*c* The comparator

*v* The value to compare against

***Returns:***

A list of users that matches the [search](#) criteria or null if no match.

**List<[User](#)> getUsersByName (String *n*)**

Finds all user that match the given name.

**Pre condition(s)**

None

**Post condition(s)**

None

***Parameters:***

*n* The name to [search](#) for

***Returns:***

A list of users that matches the [search](#) criteria or null if no match.

**List<[User](#)> getUsersByWonCompetitions ([GeneralEnum.COMPARATOR](#) *c*, int *value*)**

Find users with the number of won competitions that meets the criteria made up by the comparator *c* and the value *v* of.

**Pre condition(s)**

None

**Post condition(s)**

None

***Parameters:***

*c* The comparator

*v* The value to compare against

**Returns:**

A list of users that matches the [search](#) criteria or null if no match.

**List<[User](#)> GetUsersByNumberOfCompetitions  
([GeneralEnum.COMPARATOR](#) *c*, int *value*)**

Find users with the number of participated competitions that meets the criteria made up by the comparator *c* and the value *v*.

**Pre condition(s)**

None

**Post condition(s)**

None

**Parameters:**

*c* The comparator

*v* The value to compare against

**Returns:**

A list of users that matches the [search](#) criteria or null if no match.

**void addUser ([User](#) *u*)**

Add a new user to the database.

**Pre condition(s)**

User does not exist

**Post condition(s)**

User is in the database

**Parameters:**

*u* The user to add

**void addAvatar (long *userID*, [Picture](#) *p*)**

Add an avatar picture to the user's [profile](#).

**Pre condition(s)**

User exists

**Post condition(s)**

Users avatar/profile picture is set to *p*

**Parameters:**

*userID* ID of the user to get

*p* The avatar picture.

## static [UserAgent](#) getInstance () [static]

Returns the [UserAgent](#) instance.

It makes sure that there is only one [UserAgent](#) instance, if there is none, one is created.  
Implements the singleton design pattern.

### Pre condition(s)

None

### Post condition(s)

The singleton instance of the [GustBookAgent](#) is returned.

### *Returns:*

The [UserAgent](#) instance.

---

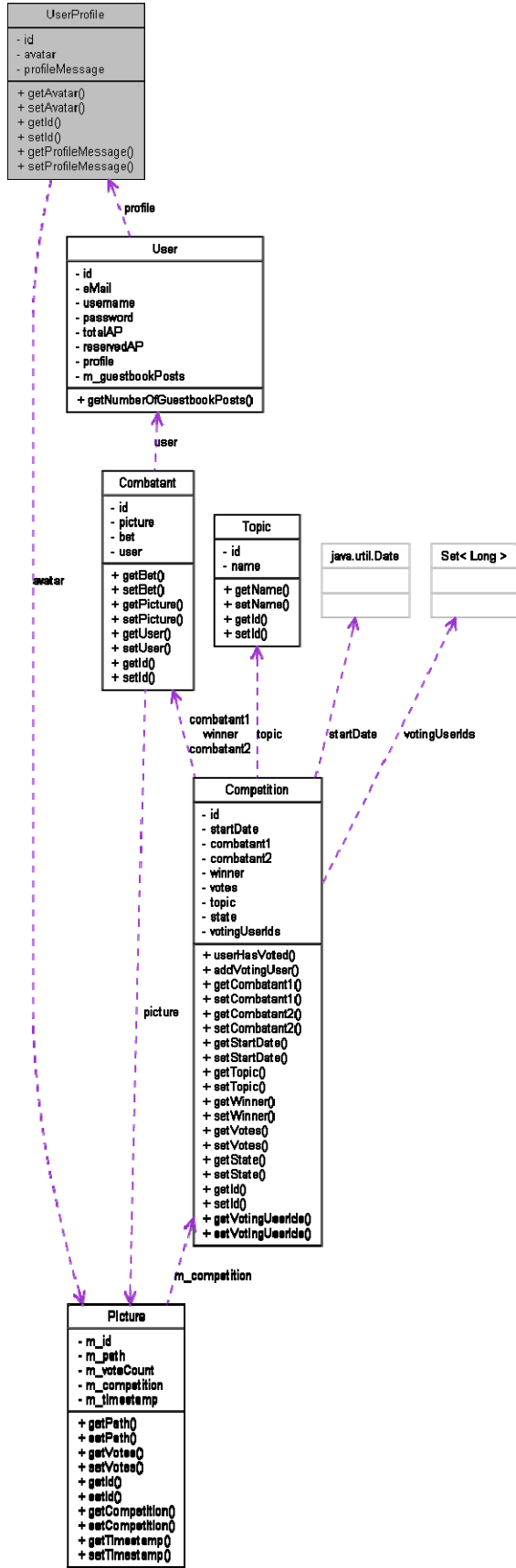
The documentation for this class was generated from the following file:

- [UserAgent.java](#)

### ***UserProfile Class Reference***

A database entity that represents a user [profile](#).

Collaboration diagram for UserProfile:





## Public Member Functions

- [Picture](#) `getAvatar ()`
  - void `setAvatar (Picture avatar)`
  - Long `getId ()`
  - void `setId (Long id)`
  - String `getProfileMessage ()`
  - void `setProfileMessage (String profileMessage)`
- 

## Detailed Description

A database entity that represents a user [profile](#).  
This class will be saved in the database using hibernate.

---

## Member Function Documentation

[Picture](#) `getAvatar ()`

void `setAvatar (Picture avatar)`

Long `getId ()`

void `setId (Long id)`

String `getProfileMessage ()`

void `setProfileMessage (String profileMessage)`

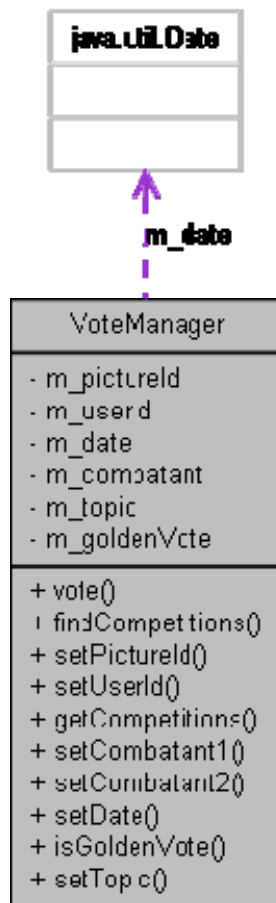
---

The documentation for this class was generated from the following file:

- [UserProfile.java](#)

## VoteManager Class Reference

Collaboration diagram for VoteManager:



## Public Member Functions

- void [vote](#) ()  
*Vote is called by voteJSP when a user votes for a picture in a competition.*
- void [findCompetitions](#) ()  
*Finds all competitions that are in the voting phase and matches the fields 'date', 'combatant1', 'combatant2', 'topic' and 'golden vote'.*
- void [setPictureId](#) (Long pictureId)
- void [setUserId](#) (long userId)
- List< [Competition](#) > [getCompetitions](#) ()
- void [setCombatant1](#) (String combatant)
- void [setCombatant2](#) (String combatant)
- void [setDate](#) (Date date)
- boolean [isGoldenVote](#) ()
- void [setTopic](#) (String topic)

---

## Member Function Documentation

### **void vote ()**

Vote is called by voteJSP when a user votes for a picture in a competition.

The method will use the two fields 'pictureId' and 'userId' as parameters. The 'pictureId' is the id of the picture that the user votes for and 'userId' is the id of the voting user.

This method will first verify that the user are allowed to vote for the picture.

### **Pre condition>**

None

### **Post condition>**

None

### **void findCompetitions ()**

Finds all competitions that are in the voting phase and matches the fields 'date', 'combatant1', 'combatant2', 'topic' and 'golden vote'.

The result is then placed in the competitions field.

### **Pre condition>**

None

### **Post condition>**

None

### **void setPictureId (Long *pictureId*)**

### **Pre condition>**

None

### **Post condition>**

None

### ***Parameters:***

*pictureId*

### **void setUserId (long *userId*)**

### **Pre condition>**

None

### **Post condition>**

None

***Parameters:***

*userId*

List<[Competition](#)> getCompetitions ()

**Pre condition>**

None

**Post condition>**

None

***Returns:***

void setCombatant1 (String *combatant*)

**Pre condition>**

None

**Post condition>**

None

***Parameters:***

*combatant*

void setCombatant2 (String *combatant*)

**Pre condition>**

None

**Post condition>**

None

***Parameters:***

*combatant*

void setDate (Date *date*)

**Pre condition>**

None

**Post condition>**

None

***Parameters:***

*date*

**boolean isGoldenVote ()**

**Pre condition>**

None

**Post condition>**

None

***Returns:***

**void setTopic (String *topic*)**

**Pre condition>**

None

**Post condition>**

None

***Parameters:***

*topic*

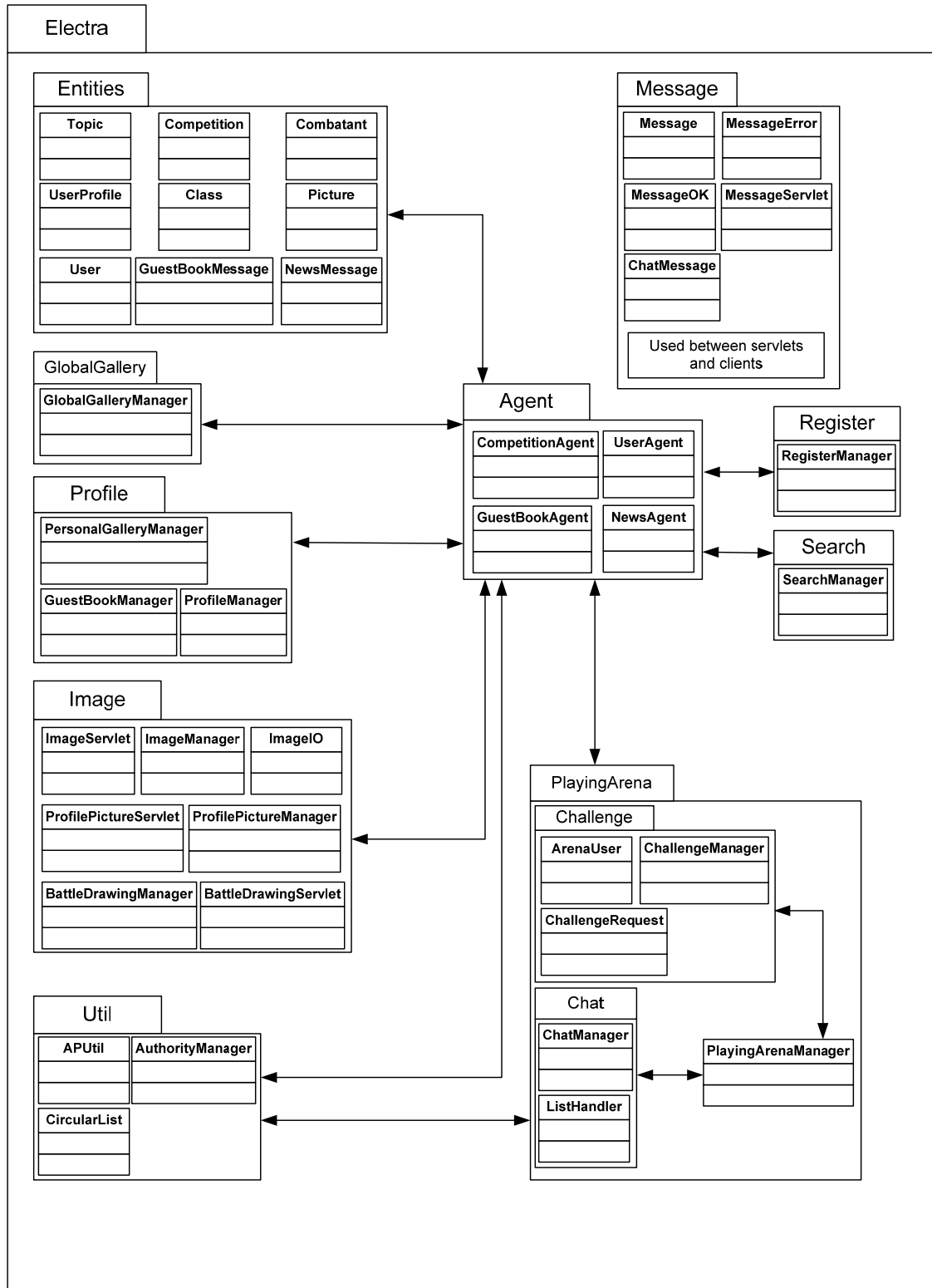
---

The documentation for this class was generated from the following file:

- [VoteManager.java](#)

- [VoteManager.java](#)

## 5.6 Package diagram



## 6. Test cases

### 6.1. Register

**Description:**

Test of the register phase.

**References:**

4.1.1.1

**Inputs:**

Username (existing, non-existing), password and email-address (valid)

**Expected output:**

- a) You can login with the new user.
- b) Error messages.

**Procedure:**

1. Go to the register page.
2. Create a new user using the non-existing username, a password of your choice and the email address.
3. Go to the mailbox and verify that you have received a mail with a a confirmation link.
4. Follow the link.
5. Go to the login page and verify a according to test case login that you can login with the new username/password.
6. Try to register with the existing username and the valid email.  
Verify that you get an error message that asks you to choose a different username.
7. Try registering with a non-valid email address (e.g. one without @)  
Verify that you get an error message saying the email is incorrect.



## 6.2. Login

### Description:

Test of the login mechanism.

### References:

FR: 4.1.1.2

### Inputs:

Username and password.

Both for an existing user and a non-existing user.

### Expected output:

- a) Redirection to profile page
- b) Error message

### Procedure:

1. Login using the existing user with correct password.  
Make sure you are redirected to the profile page of *that* user.
2. Try to login using the existing user with an incorrect password.  
You should get a message saying that the username/password pair is invalid.
3. Try to login using the non-existing user with any password.  
You should get a message saying that the username/password pair is invalid.

## 6.3. Artistic points

### **Description:**

Test of artistic points given upon registration.

### **References:**

FR: 4.1.2.1  
Appendix A

### **Inputs:**

Non-existing username.

### **Expected output:**

100 AP received upon registration.

### **Procedure:**

1. Create a new user using the non-existing username.
2. Verify that the new user has 100 AP when the profile page comes up.

## 6.4. Guestbook

### Description:

Test of guestbook. Would take forever to test every guestbook so this will be a sample test.

### References:

4.1.2.2

### Inputs:

An existing user (A). Some (at least 1) other users usernames.

### Expected output:

The contents of the guestbooks show up upon entering user profile pages, and new posts are shown perceptually immediately.

### Procedure:

1. Login with user A
2. Verify that the guestbook for the user is displayed on the profile page.
3. Post a new message and verify that it is visible immediately.
4. Repeat 1-3 with for another user's guestbook

## 6.5. Personal gallery

### **Description:**

A test of the personal gallery

### **References:**

FR: 4.1.2.3

### **Inputs:**

none

### **Expected output:**

Pictures created in battle mode visible in personal gallery.

### **Procedure:**

1. Enter a competition.
2. Before the competition ends. Make sure the picture in the competition is not visible in the personal gallery on the profile page.
3. When the competition ends, make sure that the picture drawn in that competition is visible in the personal gallery.

## 6.6. Statistics

### Description:

Test that some statistics about each user are publicly available and are available at the profile page and in the playing arena.

These statistics are:

Number of competitions participated in

Number of won competitions

Total AP

### References:

FR: 4.1.2.4, 4.1.3.6

### Inputs:

One user for checking (A) and another for usage (B).

Again, a test for all users would be too cumbersome, so a sample test is done.

### Expected output:

Consistent statistics about the user.

### Procedure:

1. Log in as user B
2. Make sure user A is signed up in the playing arena, and that A is not in a competition that is about to end (end of voting phase) during this test case.
3. Select A in the playing arena.
4. Verify that number of participated competitions, number of win competition and total AP is displayed. Make a note of these statistics.
5. Go to A's profile page.
6. Verify that the above mentioned statistics are displayed.
7. Verify that the statistics are consistent with those from the playing arena.

## 6.7. Playing arena

### **Description:**

A test that the playing arena is operational

### **References:**

4.1.3.1

### **Inputs:**

none

### **Expected output:**

Playing arena is operational.

### **Procedure:**

1. Signup (test case: )
2. Test chat room (test case: )
3. Test challenge (test case: )
4. Test competition (test case: )

## 6.8. Playing arena idle

### Description:

A test to verify that you can see some of the things that are happening in the playing arena, even if you aren't signed up. You need two computers or at least two different web browsers (from different vendors) to test this.

### References:

FR: 4.1.3.2

### Inputs:

Two users (A, B).

### Expected output:

Chat messages are delivered and signed up users are visible even if you aren't signed up.

### Procedure:

1. Make sure you aren't signed up in the playing arena with user A.
2. Make sure you ARE signed up in the playing arena with user B.
3. Enter playing arena with user A without signing up.
4. Verify from A's perspective that B is visible in the list of signed up users.
5. Post a chat message with user B.
6. Verify from A's perspective that the chat message comes up.
7. Sign out of the playing arena with B.
8. Verify from both users perspective that B is no longer visible in the list of signed up users.

## 6.9. Signup

### Description:

A test that shows that interaction with the playing arena cannot be done unless a user is signed up in it.

### References:

4.1.3.3

### Inputs:

A user (A).

### Expected output:

The user cannot interact with the playing arena unless signed up in it.

### Procedure:

1. Log in as A
2. Enter playing arena without signing up
3. Verify that you cannot challenge other users
4. Verify that you cannot post messages in the chat
5. Sign up
6. Post a message in the chat (according to test case: )
7. Challenge another user (according to test case: )



## 6.10. Show challengeable users

### Description:

Test if users that sign up in the playing arena end up in the list of challengeable users.

### References:

4.1.3.4

### Inputs:

A user (A)

### Expected output:

A end up in list of challengeable users upon signup.

### Procedure:

1. Log in with user A
2. Enter playing arena without signing up
3. Verify user A is not in the list.
4. Sign up
5. Verify that A is now in the list

## 6.11. Chat room

### **Description:**

Test the functionality of the chat room.

### **References:**

4.1.3.5

### **Inputs:**

A user (A)

### **Expected output:**

Messages posted in the chat room are actually displayed. A user cannot post messages if not signed up in the playing arena.

### **Procedure:**

1. Log in with user A
2. Enter playing arena without signing up
3. Verify that you cannot post messages in the chat room
4. Sign up
5. Post message in the chat room
6. Verify that the message is displayed.

## 6.12. Challenging users

### Description:

Test the ability to challenge other users.

### References:

FR: 4.1.3.7

FR: 4.1.6.1.1

### Inputs:

- A second user
- A time limit
- Amount of AP in percent

### Expected output:

4. A message that the challenge request has be sent to the user.
5. An error message, i.e 'The user has timed out'.

### Procedure:

1. Go to the playing arena
2. Click on the 'Signup' button
3. Select a user from the user list
4. Click on the 'Challenge' button
5. The system should respond with either (a) or (b)

## 6.13. Drawing board colors

### **Description:**

The procedure tests that a drawing can be made with different colors.

### **References:**

FR: 4.1.4.1

### **Inputs:**

- One user

### **Expected output:**

A drawing with different colors

### **Procedure:**

1. Go to the free sketch page
2. Select a color and a drawing tool
3. Draw something
4. Repeat steps 2 and 3 with a few other colors

## 6.14. Painting tools - Pencil

### Description:

Test drawing with the pencil tool.

### References:

FR: 4.1.4.2

### Inputs:

Mouse input

### Expected output:

A trace of the pencil movement on the drawing area with the selected color.

### Procedure:

1. Either
  - a. Go to the free sketch page.
  - b. Start a battle.
2. Select the pencil tool
3. Hold down the left mouse button and make some movements with the mouse on the drawing board

## 6.15. Painting tools - bucket

### Description:

Test drawing with the bucket tool.

### References:

FR: 4.1.4.2

### Inputs:

- Mouse input

### Expected output:

The point which the bucket tool was used should be filled with the selected color, outwards until some edge is reached, or until the edges of the drawing board is reached.

### Procedure:

1. Either
  - a. Go to the free sketch page.
  - b. Start a battle.
2. Select the bucket tool
3. Click somewhere on the drawing board

## 6.16. Free sketch mode

### **Description:**

The procedure tests the free sketch mode.

### **References:**

FR: 4.1.5.1

### **Inputs:**

- Mouse input

### **Expected output:**

Being able to draw on the drawing board.

### **Procedure:**

1. Go to the free sketch page.
2. Select some drawing tool and draw on the drawing board.

## 6.17. Competition

### Description:

The procedure tests the three phases of a competition.

### References:

FR: 4.1.6.1.1

FR: 4.1.6.1.2

FR: 4.1.6.1.3

### Inputs:

- Two users {A, B} for challenge and battle phase.
- At least one user {C, D, ...} different from A and B for the voting phase.

### Expected output:

A winner is announced after the voting phase has ended.

### Procedure:

1. Let user A challenge user B
2. Let user B accept the challenge
3. Both will be forwarded to the battle arena
4. Let A, B each create a drawing based on the topic
5. Once A and B are finished with the battle, their battle and drawings should show up on the Global Galley.
6. Let users C, { D, ... } vote for A and Bs drawings.



## 6.18. Voting

### Description:

The procedure tests the voting phase.

### References:

FR: 4.1.6.1.1

FR: 4.1.6.1.2

FR: 4.1.6.1.3

### Inputs:

- A competition between users A and B in the voting phase
- At least one user  $\{C, D, \dots\}$  different from A and B for the voting phase.

### Expected output:

- a. A winner is announced after a 24 hour period.
- b. A tie has occurred after a 24 hour period, goes into golden vote phase.

### Procedure:

1. Let users  $C, \{D, \dots\}$  vote for A and B's drawings during a 24 hour period after the battle phase of A and B.

## 6.19. Golden Vote

### **Description:**

The procedure tests the golden vote state.

### **References:**

FR: 4.1.6.1.1

FR: 4.1.6.1.2

FR: 4.1.6.1.3

### **Inputs:**

- A competition between users A and B which has ended in a tie
- One user C different from A and B for the voting phase.

### **Expected output:**

A winner is announced after the user C votes

### **Procedure:**

1. Let user C vote on the competition which has been ties (gold vote).

## 6.20. AP transfer

### **Description:**

The procedure tests that AP is transferred from the loser to the winner after a finished competition.

### **References:**

FR: 4.1.6.1.3

### **Inputs:**

- A competition between users A and B which has ended.

### **Expected output:**

The amount of AP reserved will be transferred from the losers to the winners total AP. The winners reserved AP will be added to his/hers total AP.

### **Procedure:**

1. Let A and B play in a competition.
2. Wait for the voting phase of the competition to finish.

## 6.21. Challenge options – Time limit

### Description:

The procedure tests that the time limit can be configured before a challenge request is made.

### References:

FR: 4.1.6.2.1

### Inputs:

- Two users A and B

### Expected output:

A battle which lasts for the selected number of minutes

### Procedure:

1. Let user A and B signup in the playing arena.
3. Let user A select a time limit
4. Let user A challenge user B
5. Let user B accept the challenge
6. For each of the users A and B
  - a. Time how long it takes for the battle to time out from time the battle starts.

## 6.22. Challenge options – Bet

### Description:

The procedure tests that the amount of AP in percent can be configured before a challenge request is made.

### References:

FR: 4.1.6.2.1

FR: 4.1.6.2.3

### Inputs:

- Two users A and B

### Expected output:

The selected amount of AP in percent (at the time the challenge was made) is transferred from the loser to the winner after the competition has ended.

### Procedure:

1. Let user A and B signup in the playing arena.
2. Let user A select amount of AP in percent.
3. Let user A challenge user B
4. Let user B accept the challenge
5. Let the competition end.

## 6.23. Challenge request

### **Description:**

The procedure tests that the user being challenged receives the challenge request

### **References:**

FR: 4.1.6.2.2

### **Inputs:**

- Two users A and B

### **Expected output:**

A challenge request is received by user B.

### **Procedure:**

1. Let user A and B signup in the playing arena.
2. Let user A challenge user B

## 6.24. AP reservation

### Description:

The procedure tests that the AP of the two user in a competition is reserved during the competition.

### References:

FR: 4.1.6.2.4

### Inputs:

- Two users A and B

### Expected output:

The reserved AP of users A and B is independently increased with amount (in percent at the time the challenge request was made) that challenger (user A) choose during the challenge. request.

### Procedure:

1. Let the users A and B be undisturbed during the procedure
2. Let user A and B signup in the playing arena.
3. Let user A select the amount of AP in percent
4. Let user A challenge user B
5. Let user B accept the challenge
6. Check that their reserved AP has increased with the correct amount during the challenge.

## 6.25. Battle topic

### **Description:**

The procedure tests that the topic of the battle is randomly selected for each battle

### **References:**

FR: 4.1.6.3.1

### **Inputs:**

- Several pairs of users

### **Expected output:**

The topic of each battle is different; some topics can be the same. But not all.

### **Procedure:**

1. Let each pair of user signup in the playing arena.
2. Let one of them challenge the other
3. Let the other accept the challenge
4. Check that the topics differ



## 6.26. Vote time limit

### Description:

Tests if a competition is no longer possible to vote for after 24 hours have passed, and that it is possible within 24 hours.

### References:

FR: 4.1.6.4.1

### Inputs:

Competition ID for one competition that ended less than 24 hours ago and one that ended more than 24 hours ago.  
Picture ID.

### Expected output:

For the competition that ended less than 24 hours ago: A message telling you that the vote has been registered.

For the competition that ended more than 24 hours ago: An error message telling you that the voting is closed.

### Procedure:

1. Make sure that you have a competition ended less than 24 hours ago (A) and one that ended more than 24 hours ago (B).
2. Vote for one of the pictures in A and make sure that the competition score is updated and the success message is displayed.
3. Vote for B and make sure that the competition score is not updated and the error message is displayed.

## 6.27. Voting page

### Description:

Tests that there is a voting page and that only competitions that are in the voting page are displayed there. Also tests that the information about the competitors are not displayed.

### References:

FR: 4.1.6.4.2, 4.1.6.4.8

### Inputs:

Competition ID for one competition that ended less than 24 hours ago and one that ended more than 24 hours ago.  
Picture ID.

### Expected output:

For the competition that ended less than 24 hours ago: An entry on the voting page displaying the pictures of that competition, but no information about the competitors  
For the competition that ended more than 24 hours ago: The competition is not shown on the voting page.

### Procedure:

1. Make sure that you have a competition ended less than 24 hours ago (A) and one that ended more than 24 hours ago (B).
2. Navigate to the voting page.
3. Make sure that A is displayed with no information about the competitors, and that B is not displayed.

## 6.28. Vote weight

### Description:

Tests that a competitor gets more points from a vote by a user (A) with more AP than from a user (B) with less AP than A.

### References:

FR: 4.1.6.4.3

### Inputs:

Competition ID for one competition that ended less than 24 hours ago.  
Picture ID.

### Expected output:

The vote casted by A gives the competitor with the picture ID more points than the vote casted by B.

### Procedure:

1. A votes for the competitor.
2. Check the difference from the amount of points the competitor had before.
3. B votes for the competitor.
4. Check the difference from the amount of points the competitor had before.
5. Make sure that the difference is larger after A's vote.

## 6.29. Vote statistics

### Description:

Tests that statistics for a vote are displayed only to users that:

- Has participated in a competition
- Has voted for that competition

### References:

FR: 4.1.6.4.4

### Inputs:

Competition ID for one competition that ended less than 24 hours ago.

User that has participated in the competition (A).

User that has voted for the competition (B).

User that has not voted for the competition (C).

### Expected output:

The statistics are only displayed to A and B.

### Procedure:

1. A enters the voting page and makes sure that the statistics are displayed.
2. B enters the voting page and makes sure that the statistics are displayed.
3. C enters the voting page and makes sure that the statistics are not displayed.

## 6.30. Voter gets AP, can only vote once and not in own competition

### Description:

Tests that a user gets AP by voting, that the user only can vote for a competition once, and that a user can not vote for a competition in which he participate.

### References:

FR: 4.1.6.4.5, 4.1.6.4.6, 4.1.6.4.7

### Inputs:

Competition ID for one competition that ended less than 24 hours ago.

User that has participated in the competition (A).

User that has voted for the competition (B).

User that has not voted for the competition (C).

### Expected output:

User A gets an error message when trying to vote for the competition and the competition score remains unchanged.

User B gets an error message when trying to vote for the competition and the competition score remains unchanged.

User C gets AP by voting for the competition.

### Procedure:

1. A tries to vote for the competition.
2. B tries to vote for the competition.
3. C tries to vote for the competition.

## 6.31. Top ten user list

### **Description:**

Tests that the users with the most AP are displayed in a list available to all users.

### **References:**

FR: 4.1.7.1

### **Inputs:**

The ten users with most AP.

### **Expected output:**

A page with the ten users listed.

### **Procedure:**

6. Navigate to the page containing the list and make sure that the users displayed matches the users from the input.

## 6.32. Global gallery

### **Description:**

Tests that the Global gallery shows all the pictures in the system.

### **References:**

FR: 4.1.7.2

### **Inputs:**

Picture ID of all the pictures in the system.

### **Expected output:**

For each picture in the system the picture is displayed in the Global gallery.

### **Procedure:**

For each picture:

1. Make sure that the specific picture ID from the database matches exactly one picture in the Global gallery.

## 6.33. Search users

### Description:

Tests that the search function returns a user if the search string matches a user in the database, and an error message otherwise.

### References:

FR: 4.1.7.3

### Inputs:

Search string matching a user in the database (A), and a string that is not (B).

### Expected output:

For A the system should send the user to the user matching string A's profile page.  
For B the system should return an error message.

### Procedure:

1. Use the search function with A as input.
2. Make sure you are directed to the profile page of user A.
3. Use the search function with B as input.
4. Make sure you get an error message.