Software Engineering Group Project
Maintenance Manual

Author:	G12
Config Ref:	SE_G12_MM_00.
Date:	2013-02-15
Version:	1.0
Status:	Release

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

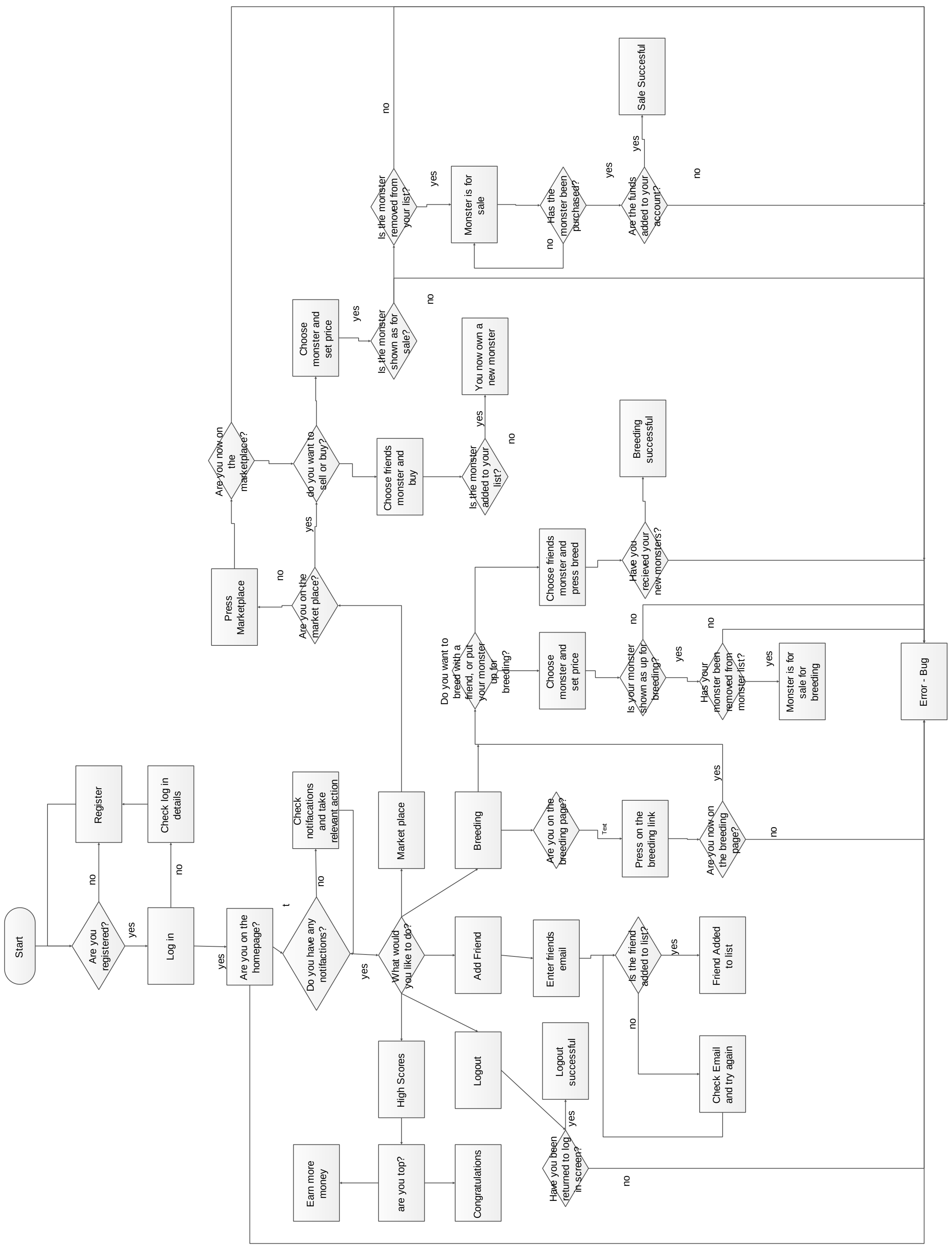
1. PROGRAM DESCRIPTION

Monster Mash is a web based game that runs on standard browsers, in which each client hosts a selection of monsters, which contain certain genetic attributes to enhance battling and breeding with other clients on different servers. Battling can be used to gain money; alternatively the players can breed and sell monsters for additional funds. Additionally, players can send and receive friend requests to interact in the game with other players. The objective of the game is to be top of the leaderboard, and to enhance knowledge of genetics whilst enjoying the game itself at the same time.

2. PROGRAM STRUCTURE

2.1 Methods

2.1.1 Flow Chart


Start
Are you registered?
yes
no
Register
Log in
no
Check log in details
Are you on the homepage?
yes
Do you have any notifactions?
t
no
Check notifacations and take relevant action
yes
What would you like to do?
Market place
Breeding
Add Friend
High Scores
Logout
are you top?
Earn more money
Congratulations
Logout successful
Have you been returned to log in screen?
yes
no
Enter friends email
Is the friend added to list?
yes
no
Friend Added to list
Check Email and try again
Are you on the breeding page?
Text
Press on the breeding link
Are you now on the breeding page?
yes
no
Do you want to breed with a friend, or put your monster up for breeding?
Choose friends monster and press breed
Have you recieved your new monsters?
Breeding successful
Choose monster and set price
Is your monster shown as up for breeding?
no
yes
Has your monster been removed from monster list?
no
yes
Monster is for sale for breeding
Error - Bug
Are you on the market place?
no
yes
Press Marketplace
Are you now on the marketplace?
do you want to sell or buy?
Choose friends monster and buy
Is the monster added to your list?
yes
no
You now own a new monster
Choose monster and set price
yes
Is the monster shown as for sale?
no
Is the monster removed from your list?
no
yes
Monster is for sale
Has the monster been purchased?
no
yes
Are the funds added to your account?
yes
no
Sale Succesful

2.1.2 Servlets

In all the servlets there are three methods that are found in all of them.

do Get(request: HttpServletRequest, response: HttpServletResponse)

doPost(request: HttpServletRequest, response: HttpServletResponse)

processRequest(request: HttpServletRequest, response: HttpServletResponse)

They are responsible for handling the incoming request to the servlet, GET request calls *do Get*, POST request calls *doPost*. In most of the application the *processRequest*, method is called from *do Get* and *doPost* and the request is handled at one place, and it does not matter if the request is a POST or GET request.

2.1.3 Monster

Detailed descriptions of the *breeding* and *fight* methods can be found below under Algorithms. [1]

2.1.4 Player

sortByMoney(players: ArrayList<Player>) : ArrayList<Player>

This methods uses bubble sort to sort the *ArrayList* of players by the amount of money. The method will sort the players from richest to poorest, so the player with the most money will be the first element in the returned *ArrayList*.

2.1.5 PersistenceManager and OtherPersistenceManager

Descriptions about the methods and the purposes can be found in Class Description of Persistence Manager and Other Persistense Manager [2]

2.1.6 RemoteTalker

findUser(userID: String) : Player

This method will attempt to find a user with a userID, it will first look locally in the applications database for the user. If the player is not found it will iterate over all the servers that are registered in the server directory and attempt to send a request to them.

getRemotePlayer(userID: String, remoteAddress: String) : Player

This method contacts a remote server to request a user. It will not search locally for the user, or try to contact all servers, only the server with the address passed in the *remoteAddress* parameter.

2.2 Program Modules

Initially we had begun with six major modules, these being Web Pages, Source Packages, Test Packages, Libraries, Test Libraries and Config Files. This then led to more sub modules being generated and created to allow the classes to be broken down further into sub modules and packages. However, Libraries, Test Libraries and Config Files were sole modules and taking Libraries and Test Libraries, only had the JAR files needed for the program to run with the different format styles. Config Files alternatively included the web.xml file and the glassfish-web.xml file to work with the Web Module and the server to be running.

The three main modules are Web Pages, Source Packages and Test Packages;

Web Pages module includes:

- WEB-INF
- CSS
- Images
- JS

WEB-INF is a module in which inside is the JSP files for all the different web pages that have been created. It stores the information of the JSP data in these classes and allows for that data to be implemented with the java and XHTML code.

CSS holds the cascading style sheets for all the web pages, which allows it to have a flowing look, and allows for styling to be generated no matter what the user decides to do.

Images contains the images used throughout the program for easy retrieval.

JS holds the data of the Jquerey and custom JS files, as well as the menu, allowing for the design to be laid out on the web page.

Source Packages module includes

- default package
- ServerCom
- data
- *database*

2.2.1 Data Package

Data module contains mainly gets and sets for the friends, monsters, player details, thus allowing the default package to work with these in creating a fully functional program.

- *CONFIG.java*
- *FightRequest.java*
- *Friend.java*
- *Monster.java*
- *NameGenerator.java*
- *Notification.java*
- *Player.java*

2.2.2 Database Package

database is the key module for our databases, it contains OtherPersistenceManager and PersistenceManager. These hold an incredibly large amount of data for communication in all forms with the user and the database, as well as working cross server to full effect. All about storing and retrieving, these are very important to allow the whole game to function.

- *OtherPersistenceManager.java*

- *PersistenceManager.java*

2.2.3 ServerCom Package

ServerCom this module is the basis for server to server communication within the program. It holds similar data to default package, but allows for the server to communicate when needed with alternative servers through the API that was created for this assignment.

- *BreedServlet.java*
- *BuyServlet.java*
- *FightLost.java*
- *FightReject.java*
- *FightRequestServlet.java*
- *FightWon.java*
- *FriendAccept.java*
- *FriendReject.java*
- *FriendRequest.java*
- *JSONManager.java*
- *RemoteTalker.java*
- *User.java*

2.2.4 Default Package

default package contains methods and algorithms for the web page which will use java code, for example to work out if an input is valid, or for ending the script on logout. It is one of the main modules used in our program and allows for all of the client side data to be produced when the user is playing the game.

- *CreateAccountPage.java*
- *FightAccept.java*
- *FightingPage.java*
- *HighscoresPage.java*
- *LoginPage.java*
- *LogoutPage.java*
- *MarketPage.java*
- *MatingPage.java*
- *Unregister.java*

For further details see Class descriptions found in the Design Spec under Class Descriptions. [3]

2.3 Algorithms and Methods

2.3.1 CreateAccountPage.java

```
private boolean isValidEmailAddress(String email) {
        boolean result = true;
        try {
            InternetAddress emailAddr = new InternetAddress(email);
            emailAddr.validate();
        } catch (AddressException e) {
            result = false;
        }
        return result;
    }
```

This method verifies that the email entered into the register page is a true email, and it also tests the email address to check whether or not it will validate. This will return the result false, if it fails, and will continue if it is true.

2.3.2 HighscoresPage.java

```
private void getDataFromDB(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        HttpSession session = request.getSession(false);
        if(session == null || session.getAttribute("user") == null){
            // Redirects when user is not logged in
            response.sendRedirect("");
        }else{
            Player current = (Player)session.getAttribute("user");
            PersistenceManager pm = new PersistenceManager();
            // Updates player informations
            current = pm.getPlayer(current.getUserID());
            session.setAttribute("user", current);
            // Saves all notifications to attribute
            request.setAttribute("notificationList", current.getNotifications());
            // Saves all friends and friend requests to attribute
            request.setAttribute("friendList", current.getFriends());
            // Saves all friend requests to attribute
            request.setAttribute("requestList", pm.getFriendRequestList(current.getUserID()));
            // Saves all monsters to attribute
            request.setAttribute("monsterList", pm.getMonsterList(current.getUserID()));
            request.getRequestDispatcher("/WEB-INF/highscores_page.jsp").forward(request, response);
        }
    }
```

This method gets the data from the database through a HTTP servlet request, it will redirect when user isn’t logged in, however will save the users data to the database when logged in. It gets the players information, saves all attributes and monsters, and finally saves the monster list to the user ID in the highscores.jsp file.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
        HttpSession session = request.getSession(false);
        if(session == null || session.getAttribute("user") == null){
            // Redirects when user is not logged in
            response.sendRedirect("");
        }else{
            PersistenceManager pm = new PersistenceManager();
            Player current = (Player)session.getAttribute("user");
            ArrayList<String> highscores = pm.getHighscores(current.getUserID());
            request.setAttribute("highscores", highscores);
            this.getDataFromDB(request, response);
        }
```

```
   }
```

Similar to before, however this actually requests the highscores and responds the database data into a table format to display the information.

2.3.3 MainPage.java

```
private void sendFriendRequest(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        HttpSession session = request.getSession(false);
        // Checks if user is logged in
        if(session != null && session.getAttribute("user") != null){
            // Gets email from POST
            String email = request.getParameter("email");
            PersistenceManager pm = new PersistenceManager();
            Player sender = (Player)session.getAttribute("user");
            // Checks if user with this email address exists
            String[] receiver = pm.getPlayerIdAndServer(email);
            if(receiver[0].equals("0")){
                request.setAttribute("alertMessage", "Cannot find user with this email address.");
            }else if(pm.isFriendRequestSent(sender.getUserID(), receiver[0])){
                request.setAttribute("alertMessage", "Cannot send friend request to this player.");
            }else if(sender.getUserID().equals(receiver[0])){
                request.setAttribute("alertMessage", "Cannot send friend request to yourself.");
            }else{
                String message = "Friend request to <b>"+email+"</b> sent successfully.";
                sender.addNotification(new Notification(message, "You have sent friend request to
<b>"+email+"</b>.", sender));
                pm.storeNotifications(sender);
                int receiverServerID = Integer.parseInt(receiver[1]);
                pm.sendFriendRequest(sender.getUserID(), receiver[0], receiverServerID);
                if(receiver[1].equals("12")){
                    //Receiver is on our server
                    Player receiverObject = pm.getPlayer(receiver[0]);
                    receiverObject.addNotification(new Notification("Received friend request from
<b>"+sender.getUsername()+"</b>", "You have received friend request from
<b>"+sender.getUsername()+"</b>.", receiverObject));
                    pm.storeNotifications(receiverObject);
                    // Save updated player's object in session
                    session.setAttribute("user", sender);
                    request.setAttribute("alertMessage", message);
                }else{
                    // TODO: Receiver is on different server
                }
            }
        }
   }
```

The point of this method is to check the HTTP servlet to run the PersistenceManager and receive the PlayerID and Sever (thus with the email ID). Additionally it requests the Attributes of the friend, and returns a non-valid email error message if false, along with if you send the request to yourself, another error message will appear. If the friend request is successful it will add a new notification message saying it has been success with the email in the JavaScript generate message. If it is successful, the player will receive the friend object and display in the friends list. Alternatively if you receive a request, it will show another notification stating that someone has added you, and similarly if accept the request, then it will display the friend object in the friends list.

```
private void respondToFriendRequest(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        HttpSession session = request.getSession(false);
        // Check if user is logged in
        if(session != null && session.getAttribute("user") != null){
            Player logged = (Player)session.getAttribute("user");
            PersistenceManager pm = new PersistenceManager();
            // Check if acceptFriendRequest occured:
            if(request.getParameter("acceptFriendRequest") != null){
                String friendshipID = request.getParameter("acceptFriendRequest");
                pm.acceptFriendRequest(friendshipID, logged.getUserID());
            }
            // Check if cancelFriendRequest occured:
            if(request.getParameter("cancelFriendRequest") != null){
                String friendshipID = request.getParameter("cancelFriendRequest");
                pm.cancelFriendRequest(friendshipID, logged.getUserID());
            }
        }
    }
```

As an addition to the previous method, this method checks if the response of the friendship request was successful, or it was cancelled. This gives the option to cancel a request for friendship, or to check if the friendship was successful.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
        HttpSession session = request.getSession(false);
        if (session == null || session.getAttribute("user") == null) {
            // Redirects when user is not logged in
            response.sendRedirect("");
        } else {
            PersistenceManager pm = new PersistenceManager();
            Player current = (Player) session.getAttribute("user");
            // Check if user wants to cancel offer
            this.cancelOffer(request, response, pm, current);
            // Check if user wants to buy monster
            this.buyMonster(request, response, pm, current);
            ArrayList<Monster> monsters = pm.getMonstersForSale(current.getUserID());
            // Prepare strings:
            ArrayList<String> monstersForSale = new ArrayList<String>();
            for(Monster m: monsters){
                monstersForSale.add("<li><a
href=\"market?monster="+m.getId()+"&server="+m.getServerID()+"\"><b>Name:</b> "+m.getName()+" |
<b>Owner:</b> "+pm.getPlayerUsername(m.getUserID(), m.getServerID())+" | <b>Price:</b>
"+m.getSaleOffer()+"$ | <b>Stats:</b> def: "+(int)(m.getCurrentDefence()*100)+" /  hp:
"+(int)(m.getCurrentHealth()*100)+" / str: "+(int)(m.getCurrentStrength()*100)+" </a></li>");
            }
            request.setAttribute("monstersForSale", monstersForSale);
            this.getDataFromDB(request, response);
        }
    }
```

For the buying monsters method, like other methods it begins with the session being active, if true, it has the option to check the player has cancelled their offer. Then we look at buying the monster, which goes into the monster array, which has been created from the monsters for sale method, which returns the monsters that are in fact for sale. By getting the server ID, owners' name, player Username, ID and details of the monster, the monster is then bought, and requests the monster for sale if false.

```
    private void cancelOffer(HttpServletRequest request, HttpServletResponse response, PersistenceManager pm,
Player current) throws ServletException, IOException {
        String monsterID = request.getParameter("cancelOffer");
        if(monsterID != null){
            if(pm.cancelMonsterOffer(current.getUserID(), monsterID)){
                current.addNotification(new Notification("You have canceled your offer of 
<b>"+pm.getMonsterName(monsterID)+"</b>.", "<b>"+pm.getMonsterName(monsterID)+"</b> offer has 
been canceled by you. Now offer will not appear on the market.", current));
                pm.storeNotifications(current);
            }
        }
    }
```

The method above is the cancelOffer method, which when the cancelation is true, will set the notification to show that the offer has in fact been cancelled, and the offer will no longer appear on the market.

```
    private void buyMonster(HttpServletRequest request, HttpServletResponse response, PersistenceManager 
pm, Player current) throws ServletException, IOException {
        String monsterID = request.getParameter("monster");
        String server = request.getParameter("server");
        if(monsterID != null && server != null){
            try{
                String message = null;
                int serverID = Integer.parseInt(server);
                if(!pm.monsterExists(monsterID, serverID)){
                    message = "Monster doesn't exists";
                }else if(!pm.canUserBuyMonster(current.getMoney(), monsterID, serverID)){
                    message = "You do not have enough money for buying this monster.";
                }else{
                    pm.buyMonster(current.getUserID(), monsterID, serverID);
                    message = "You have bought new monster called "+pm.getMonsterName(monsterID)+".";
                    current.addNotification(new Notification("You have bought new monster called 
<b>"+pm.getMonsterName(monsterID)+"</b>.", "You have bought new monster called 
<b>"+pm.getMonsterName(monsterID)+"</b>. It will appear on your monster list now.", current));
                    pm.storeNotifications(current);
                }
                request.setAttribute("alertMessage", message);
            }catch(Exception e){

            }
        }
    }
```

The method buyMonster firstly checks to see if a monster actually exists, and returning that fact in a message if false, similarly, if the user cannot afford the monster they are trying to buy then this will appear as a message as well. If all is well, and true, then the current monster selected from the array will be purchased and displayed as a notification on the homepage, which is requested from the servlet.

```
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, 
IOException {
        HttpSession session = request.getSession(false);
        if (session == null || session.getAttribute("user") == null) {
            // Redirects when user is not logged in
            response.sendRedirect("");
        } else {
            // Make new offer:
            Player current = (Player) session.getAttribute("user");
            String monsterID = request.getParameter("monsterID");
            String offerAmount = request.getParameter("offerAmount");
            String error = null;
            PersistenceManager pm = new PersistenceManager();
        if(monsterID == null || offerAmount == null){
            error = "Please fill both fields.";
        }else if(monsterID.length() < 1){
            error = "Please select monster name.";
        }else if(offerAmount.length() < 1){
            error = "Please specify your offer amount.";
        }else{
            int amount = 0;
            try{
                amount = Integer.parseInt(offerAmount);
                if(!pm.makeNewMarketOffer(current.getUserID(), monsterID, amount)){
                    error = "Incorrect monster name.";
                }
            }catch(Exception e){
                error = "Incorrect amount.";
            }
        }
        if(error != null){
            request.setAttribute("alertMessage", error);
        }else{
            current.addNotification(new Notification("You offered
<b>"+pm.getMonsterName(monsterID)+"</b> for sale for <b>"+offerAmount+"$</b>.",
"<b>"+pm.getMonsterName(monsterID)+"</b> is now available for sale for <b>"+offerAmount+"$</b>. You
cannot use this monster until you cancel your offer.", current));
            pm.storeNotifications(current);
            request.setAttribute("alertMessage", "You offered "+pm.getMonsterName(monsterID)+" for sale for
<b>"+offerAmount+"$</b>.");
        }
        doGet(request, response);
    }

  }
```

The doPost method is a method that allows the player to offer his monster up for sale, requiring the monsters name and a price tag. This is done by setting the user, monster and offer attributes, with constraints of monster ID <1 and amount <1, if either of these two parameters are true, the program will throw an error. If the new market offer is successful, it will return a notification providing the previous does not equal null.

2.3.4 PersistenceManager

```
private String randomString(int length){
        Random random = new SecureRandom();
        String letters = "abcdefghjkmnpqrstuvwxyzABCDEFGHJKMNPQRSTUVWXYZ23456789";
        String pw = "";
        for (int i=0; i<length; i++){
            int index = (int)(random.nextDouble()*letters.length());
            pw += letters.substring(index, index+1);
        }
        return pw;
    }
```

This method is the generator for the random monster and player names, it generates a random name, using the string letters and does a for loop to generate a selection of letters placed together. It then returns the result.

```
public boolean accountExists(String userID){
        int count = 0;
        try{
            Statement stmt = connection.createStatement();
            stmt = connection.createStatement();
            ResultSet results = stmt.executeQuery("SELECT count(\"id\") FROM \"Player\" WHERE \"username\"
= '"+userID+"'");
            results.next();
            count = results.getInt(1);
            results.close();
            stmt.close();
        }catch (SQLException sqlExcept){
            this.error = sqlExcept.getMessage();
        }
        // TODO: Check other servers! (SERVER<->SERVER)
        if(count > 0){
            return true;
        }
        return false;
    }
```

booleanaccountExsists method is a true or false return that checks to the connection in the stmt, which then links into the database to find userID, which then counts the results and returns if the account exists.

```
public ArrayList<Player> getFriendList(String playerID){
        ArrayList<Player> friendList = new ArrayList<Player>();
        try{
            Statement stmt = connection.createStatement();
            ResultSet result = stmt.executeQuery("SELECT * FROM \"Friendship\" WHERE (\"sender_id\" =
'"+playerID+"' OR \"receiver_id\" = '"+playerID+"') AND \"confirmed\" = 'Y'");
            while(result.next()){
                if(result.getString("sender_id").equals(playerID+"")){
                    // Sender String id, String name, int serverID
                    friendList.add(new Player(result.getString("receiver_id"),
this.getPlayerUsername(result.getString("receiver_id"), result.getInt("receiver_server_id")),
result.getInt("receiver_server_id")));
                }else{
                    // Receiver
                    friendList.add(new Player(result.getString("sender_id"),
this.getPlayerUsername(result.getString("sender_id"), result.getInt("sender_server_id")),
result.getInt("sender_server_id")));
                }
            }
            result.close();
            stmt.close();
        }catch (SQLException sqlExcept){
            System.err.println("Selecting friendships from DB error:\n"+sqlExcept.getMessage());
            this.error = sqlExcept.getMessage();
        }
        return friendList;
    }
```

This method works to return the friendslist, by creating a friendlist array in the players ID and dragging it across from the database in the friendship column and then returns the results through the user and server ID whilst adding the friends into the array. Both connections then close, with error messages is anything returns false, thus finally returning the friendlist array into object format on the GUI. This is the same for notifications, friendship requests and returning the monster list, just from different columns in the database.

```
public void sendFriendRequest(String senderID, String receiverID, int receiverServerID){
        try{
            Statement stmt = connection.createStatement();
            String id = this.randomString(16);
            stmt.execute("INSERT INTO \"Friendship\" (\"id\", \"sender_id\", \"receiver_id\", \"sender_server_id\", 
\"receiver_server_id\", \"confirmed\") VALUES ('"+id+"', '"+senderID+"', '"+receiverID+"', 12, 
"+receiverServerID+", 'N')");
        }catch(SQLException sqlExcept){
            System.err.println(sqlExcept.getMessage());
            this.error = sqlExcept.getMessage();
        }
    }
```

Alternatively the sendFriendRequest connects to the database in the same way as before, however it inserts into the database using a SQL statement and confirms the values of the sender and receiver ID, whilst at the same time checking for system errors and SQL exceptions. This is the same concept for receiving a friendrequest and accepting/denying a request.

```
public ArrayList<String> getHighscores(String playerID){
        ArrayList<Player> friends =  this.getFriendList(playerID);
        ArrayList<String> friendIDs = new ArrayList<String>();
        ArrayList<String> toReturn = new ArrayList<String>();
        for(Player p: friends){
            if(p.getServerID() == 12){
                friendIDs.add(p.getUserID());
            }else{

            }
        }
        friendIDs.add(playerID);
        // Preparing query
        String query = "SELECT * FROM \"Player\" WHERE ";
        for(String s: friendIDs){
            query += "\"id\" = '"+s+"' OR ";
        }
        query = query.substring(0, query.length()-4);
        query += "ORDER BY \"money\" DESC";
        try{
            Statement stmt = connection.createStatement();
            ResultSet result = stmt.executeQuery(query);
            int i = 1;
            while(result.next()){

toReturn.add("<tr><td>"+i+".</td><td><b>"+result.getString("username")+"</b></td><td>"+result.getInt("mo
ney")+"$</td></tr>");
                i++;
            }
            result.close();
            stmt.close();
        }catch (SQLException sqlExcept){
            System.err.println(sqlExcept.getMessage());
            this.error = sqlExcept.getMessage();
        }
        return toReturn;
    }
```

The getHighscores array method begins by generating 3 arrays and connecting to the server ID, thus receiving friends ID and adding the player ID from the player column in the database. It is then ordered by amount of money descending. Finally the query is looping until all results are returned into HTML table. The return gets the strings of the username and the int of money which after ordering gets closed along with the stmt and is returned to the final array to display in the HTML table form. This is similar to the monsterforbreeding method and monsterforsale method, which both connect into different columns of the database.

2.3.5 Algorithms

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
        String email = request.getParameter("email");
        String password = request.getParameter("password");
        if(email.length() < 1 || password.length() < 1){
            request.setAttribute("errorMessage", "Please enter both your email and password.");
            request.getRequestDispatcher("/WEB-INF/login_page.jsp").forward(request, response);
        }else{
            PersistenceManager pm = new PersistenceManager();
            password = this.MD5(password);
            Player selected = pm.doLogin(email, password);
            if(selected != null){
                // If player exists save object to the session called "user"
                HttpSession session = request.getSession(true);
                session.setAttribute("user", selected);
                response.sendRedirect("main");
            }else{
                // If null, there's no player with this email and password
                request.setAttribute("errorMessage", "Password or email address is incorrect.");
                request.getRequestDispatcher("/WEB-INF/login_page.jsp").forward(request, response);
            }
        }
    }
}
```

Fighting – when a fight request is accepted, fight method is being called taking opponent (Player object) as a parameter. Opponent's and player's current health values are being decreased in a loop until one of the values reaches 0. First, from opponent's current health result of multiplying player's current strength, complement of opponent's current defence and a random value between 0 and 1 is being subtracted, then from player's current health result of multiplying opponent's current strength, complement of player's current defence, and a random value between 0 and 1 is being subtracted. Once one of the values reaches 0, opponent's current health is being returned. If the returned value is larger than 0, the fight is won by the opponent, and player gets message about losing. Otherwise, the fight is lost by the opponent and player gets message about winning and gets money reward.
Breeding – number of children being result of breeding is calculated by multiplication of square root of product of both parents' fertilities and constant indicating maximum number of children. Date of birth of each child is current date and date of death is current date increased by constant indicating lifespan of a monster. Owner's ID of each child is assigned to player's ID, and name is generated using the name generator. For defence, strength, health, and fertility of each child, there is 5% chance of mutation (so it gets set to a random value between 0 and 1), and equal chance of inheriting the value from one or the other parent.
Monster's name generation – name for a monster is generated by randomly choosing a string from an array of 25 names, and randomly choosing a string from an array of other 19 names, giving 475 unique combinations.
Sorting high scores – this is achieved by applying bubble sort algorithm in sortByMoney() method. If a player with higher place on the list has less money than the next one, the places on the list are being swapped.

2.4 Main data areas

The main data structure for the system is the classes found in the data package, they store the information about our users and monsters. The data is only found as objects when needed to process a request, for most of the time the data will not be held in the program, but stored in the database and fetched when needed.
This makes the PersistenceManager and OtherPersistenceManager very important classes, since they hold the methods used to fetch and store data from the database. See the Class Descriptions section in the design specification document for more information about the specific classes. [4]

2.5 Files

The files in the web directory are needed to style and display the HTML. The directory also contains the JavaScript files and images used in HTML, so they need to be present to make the application work properly. The web directory also has the Glassfish configuration file, used to register servlets with the application.

2.6 Interface Description

Upon opening the project a user is shown the login in screen, here there is two fields in which a user will enter their email address and password and then click the sign in button to go to the Mainpage. There is also a create an account button which takes you to another page with several fields, here a user enters the details required for the to make a new account, then they can press the create an account button to go back to the login page. Once logged in a user is presented with the Mainpage, in the middle of this page a list of notifications are presented to the user, the short messages can be clicked on to provide greater detail. On the right side of the screen is the Monster Lister showing the monsters that the user owns, by clicking on the monsters in this list a display of the attributes is shown, clicking again closes this display. On the left side of this screen is the Friends List side bar, here is where the list of friends is displayed. At the bottom of this sidebar is a field that the user can enter emails into so they can send them friend requests. Friend requests appear in the friends list and a user can either accept or deny the request by clicking on it. A user can also interact with their friends using this list by clicking on them, they can then choose to either remove them as a friend or send a battle request. The side bars are shown on all the pages of the website
Above these sections of the Mainpage is a navigation bar, with a buttons that take a user to the Home, Market, Mating and Highscores pages. On the market page the notification area is replaced by the Market menu, here a list of the monsters that friends of the user have for sale is displayed. A user can buy these monsters by clicking on them in the list. Below this is a list of all the monsters that the user is currently offering for sale. Below this there is a drop down list where a user can select one of the monsters and set an offer value, thus putting that monster up for sale.
The mating page has a similar display to the market page, first showing all the monsters that the users' friends are offering for breeding, but this also includes a list in which the user selects which one of the monsters they want to breed with. Below this is the list of the monsters the user is offering for breeding, and below that the options to offer their monsters for breeding.
On the Highscores page a list of the users' friends ordered by their wealth is shown

2.7 Possible Improvements

2.7.1 AJAX

Currently the web pages has to be refreshed manually by the user for a new notification or fight/friend request to appear, a major improvement would be a AJAX system were the user would not have to refresh the page. With AJAX we could implement a system that allows sending the notifications to the users' client without the user needing to interact with the page.

2.7.2 Cross server communication

The server to server API should be extended to allow extra data, such as a Monsters name. The structure of it should also be improved, now it is very slow since a query has to be made to all servers so check if a user exists. The speed could have been improve on our side by threading our application, and setting up a remote data cache that stores data that has been requested before. There are also some security issues with the current API.

2.7.3 Site security

There are several security concerns in the system. One of the major ones is the lack of sanitization on user input from the website, and some pages rely solely JavaScript for input validation. Parts of the system is doing some form of sanitization on input, but only for SQL so the site is still vulnerable to cross site scripting and html injection at the sanitized inputs. The server to server API currently has no authentication or data verification, so requests could come from malicious servers, and request can be used to accept friend and fight request that has not been approved by the user.

2.8 Things to watch when making changes

There are no parts of the code base containing anything out of the ordinary. Many places in the code there will be references to a sender and receiver, changing the names of those variables could lead to confusion and that naming style should be consistent throughout the code base.

2.9 Limitations

2.9.1 Physical limitations of the system

One of Glassfish's major features for big enterprise applications is its scalability. New servers can be added and setup to run instances of our application, and if one of our servers in the cluster were to fail the Glassfish software would ensure that the application would still run on the remaining servers. Glassfish has support for several different platforms, including all major platforms [1]. Finding a vendor with the sells the supported platform should not be a problem, and there will be a big range of vendors to choose from. Glassfish is not a light system, according to the official documentation on Oracle's webpages [1] the minimum memory for Glassfish on all platforms is 1 GiB, and 2 GiB recommended memory. This means that the application will need at least a server with 2 GiB memories to ensure smooth performance for our users.
Hard disk is not a big problem for our application, Glassfish itself needs less than a 1 GiB of space [1], and the user data we store does not require a lot of disk space. Disk space is also one of the easiest hardware updates we can do to our system. Due to the server to server API, our system is also affected by the performance of the remote systems we are working with. If a response to a request takes a long time it will affect our performance, since the thread started to serve the request that on our side will be left hanging waiting for a response, and in some cases it will need to contact several remote servers to get the needed data. This will affect the smoothness of the application from the users' point of view.

2.9.2 MonsterMash Requirements

Operating System
Linux: Red Hat Enterprise Linux 5.0, Red Hat Enterprise Linux 4.0, SUSE Linux Enterprise Server 10, Ubuntu Linux 8.04
Windows: Windows 7 Professional, Windows XP Professional SP3, Windows 2008, Windows Vista Business
Mac OS: Mac OS X 10.5 and 10.6
Solaris: Solaris 10 (SPARC Platform), Solaris 10 (x86 Platform), OpenSolaris 2009.06
Memory
Minimum Memory: 1 GiB
Recommended Memory: 2GiB
Minimum Disk Space
Minimum disk Space: 500 MiB free
Recommended Disk Space: 1 GiB free
References
[1] Sun Glassfish Enterprise Server v3 Release Notes
http://docs.oracle.com/cd/E19226-01/820-7688/abpak/index.html

2.10 Known Bugs

2.10.1 Adding friends

1. When adding a friend that is not registered on the local server or a remote server the friend will not be found, and the system will show a HTTP error 500 page with an exception. This can be fixed by doing error checking on the returned Player object from the *findUser()* method in *PersistenceManager.*
2. The input field for a friends email on the main page is sensitive to illegal characters, and if one is entered the user will be sent to a HTTP error 500 page with an exception. This can be fixed by sanitizing the input from the form.
3. No notification when your friend request has been accepted. This can be fixed by adding a notification to the user when the request is accepted.
4. When rejecting a friend request the notifications are wrong for sender and receiver. The sender gets "You declined the friend request from receiver@test.com" and receiver gets "sender@test.com has rejected your friend request. ", and it should have been the other way around. This can be fixed by switching the notifications.

2.10.2 Fighting

1. When rejecting a fight request to a user that is on a remote server the system will show a HTTP error 500 page with an exception. This can be fixed by checking on the fight page if the sender server id in the fight request is local or not, if it is local handle it internally, if not send a request to the remote server. Currently it will only check on local server.

2.10.3 Server to server communication

1. Remote users are not stored in the database, so the user information will need to be retrieved from the remote server when needed. This makes the our site slow since it has to wait for a remote response. There is no easy fix for this, due to limitations in the API, and the fact that you have to look up data externally.
2. When a remote server gives a bad response some pages are prone to redirect the user to a HTTP error 500 page with an exception. To fix this the conditions of when this is happening has to be tested better.

2.11 Rebuilding and Re-Testing Suggestions

All files can be found in a private repository on the GitHub. Files are divided into web pages, source files and tests. All JSP pages, images, CSS and JavaScript files can be found inside "web" folder. Source files are divided into 6 packages:

- **Default package:** contains all JavaServlets (without Servlets for server to server communication).
- **ServerCom:** contains Servelets for server to server communication and RemoteTalker class, which has all methods used for communication between servers.
- **Data:** contains all model classes including Config class.
- **Database:** contains two Persistence Managers used for communication between website and database.
- **Org.json:** contains JSON library downloaded from official JSON website.

There are also two test packages containing JUnit test for all classes inside "data" and "database" packages. In JUnit tests we specify the input and expected output if input equals expected output it has passed the tests. New test are added to JUnit test files with the same name as class, where problem was discovered. All documents are in a non-standard format (PDF converted from DOC), but if we could rewrite them, we will use LaTeX template for all documents.

3. REFERENCES

[1] Maintenance Manual: Algorithms. Group 12. SE_G12_MM_00. 1.3 Release
[2] Design Specification: Interface Description. Group 12. SE_G12_DS_00. 1.2 Release
[3] Design Specification: Interface Description. Group 12. SE_G12_DS_00. 1.2 Release
[4] Design Specification: Interface Description. Group 12. SE_G12_DS_00. 1.2 Release

4. DOCUMENT HISTORY

Version	*CCF No.*	*Date*	*Changes made to document*	*Changed by*
1.0	N/A	2013/02/15	Implementation of all major elements	G12

Version	*CCF No.*	*Date*	*Changes made to document*	*Changed by*