Software Engineering Group Project
Design Specification

Author:	Group 12
Config Ref:	SE_G12_DS_00
Date:	2012-12-04
Version:	1.1
Status:	Release

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this Document

This document shows a detailed design of our game, including in depth class analysis, class diagrams and mapping requirements. It is to adhere to the design requirements [1] and the design specification requirements [2] QA documents, and to allow coders to understand and map out the classes that will be used during the creation of Monster Mash. The descriptions and diagrams are included to aid in the development of game and also to help in the maintenance of the project once it is completed

1.2 Scope

This document gives access to the design information and class information. It will explain how these are linked together and which each class does. There is also discussion of the different applications in the project and the way in which the functionality of the project will be achieved.

1.3 Objectives

- To accurately describe the classes that will be used in the project
- To show the functionality of these classes and how this will be achieved
- To create a document that aid in the creation of the prototype software
- Allow for the implementation to be completed adhering to the user requirements and efficiently

2. DECOMPOSITION DESCRIPTION

2.1 Applications in the System

There are two distinct applications - the client program, which runs in a web browser on the player's computer, and the server program, which will run on a server in the University.

Our client program will be what the player will see in the web browser. With this application, the player will be able to send friend requests, fight requests and other interactions with the server to be passed along to another player or other places, depending on the action. They can sell their monsters, breed their monsters with other players, and have their monsters fight with other players' monsters.

The server will generate the pages that are handed to the client program (the web browser). It will also pass data from the database to the users when required - such as the data from the marketplace, the leader board and the breeding market. Actions in the client program, such as selling a monster, will be passed back to the database on the server via the PersistenceManager.

Servers will also interact with each other, by hosting cross-server fights between players, trades and friend requests.

2.2 Significant Classes

2.2.1 Player

This class contains information about a single player/account. Username and password instance variables are required for signing in. To make the application more secure, the password will be encrypted. We will use the user's email address to confirm their account. For each player we will store their wealth as an integer. Each player will also have a list of friends and we decided to store them in an ArrayList of Players, because friend list will not be fixed size. There is an ArrayList of Monsters, which holds all monsters attached to a single player. Player class contains two methods. **createInitialMonster()** creates first random monster and adds it to ArrayList of monsters. **addFriend()** takes object of Friend class and adds it to the friends ArrayList.

2.2.2 Monster

Monster is aclass containing information about a single monster. It is an abstract class that is used by Male class (for male monsters) and Female class (for female monsters).
Monster class contains following attributes:
#id:int - ID of the monster
#name:String - name of the monster
#dob:Date - monster's date of birth
#genetic_stength:float - strength of the monster, used during breeding
#strength:float - strength of the monster, used during fighting
#speed:float - speed of the monster
#accuracy:float - accuracy of the monster
#endurance:float - endurance of the monster
#armor:float - the strength of monster's armor
#dodge:float - ability to dodge
#age_rate:float - age rate of the monster, value in range 0..1
#fertility:float - fertility of the monster
#health:int - health of the monster, value in range 0..100

Male is aclass containing information about a single male monster. It extends the Monster abstract class.
Male class contains following attributes:
-injured:boolean - true if monster is injured, otherwise false
Male class contains following methods:
+fight(opponent:Male):boolean - contains fighting algorithm. Takes the opponent monster as a parameter. Returns true if the monster starting the fight has won, otherwise returns false.

Female is a class containing information about a single female monster. It extends the Monster abstract class.
Female class contains following attributes:
-MAX_MAXCHILDREN:final int - maximum number of children possible to achive in single breeding. This value is the same for every female.
Female class contains following methods:
+breeding(monster:Male):Monster[] - contains breeding algorithm. Takes male monster as a parameter. Returns array of new monsters that are the result of breeding.

2.2.3 Persistence manager

This is the only class which interacts with database. It has five private final instance variables: **dbname**, **dbhost**, **dbport**, db**username**, **dbpassword** and also two private instance variables: **connection** (which holds database connection) and **error** (error message if any occurred). The **constructor** of this class, which has no parameters, opens a connection to the database. **select()** method takes a SQL query from a parameter as a string and does select query then if successful it will return array of strings otherwise return false. Next method is **query()**. It takes SQL query as a string and does SQL query like UPDATE or INSERT then it will return true if successful or false when failed. To get an error message there is simple **getError()** method, which returns string with an error message, if there are no errors this method returns null.

2.3 Requirements mapping

Functional Requirement	Classes Providing Requirement
FR1	Register, Player, Login, PersistenceManager
FR2	Player, PersistenceManager
FR3	Monster, Player, PersistenceManager
FR4	Monster, Player, FightRequest, PersistenceManager
FR5	
FR6	Register, Player, FriendRequest, Market, MarketOffer, BreedingMarket, BreedingOffer, *Monster*, MainPage,

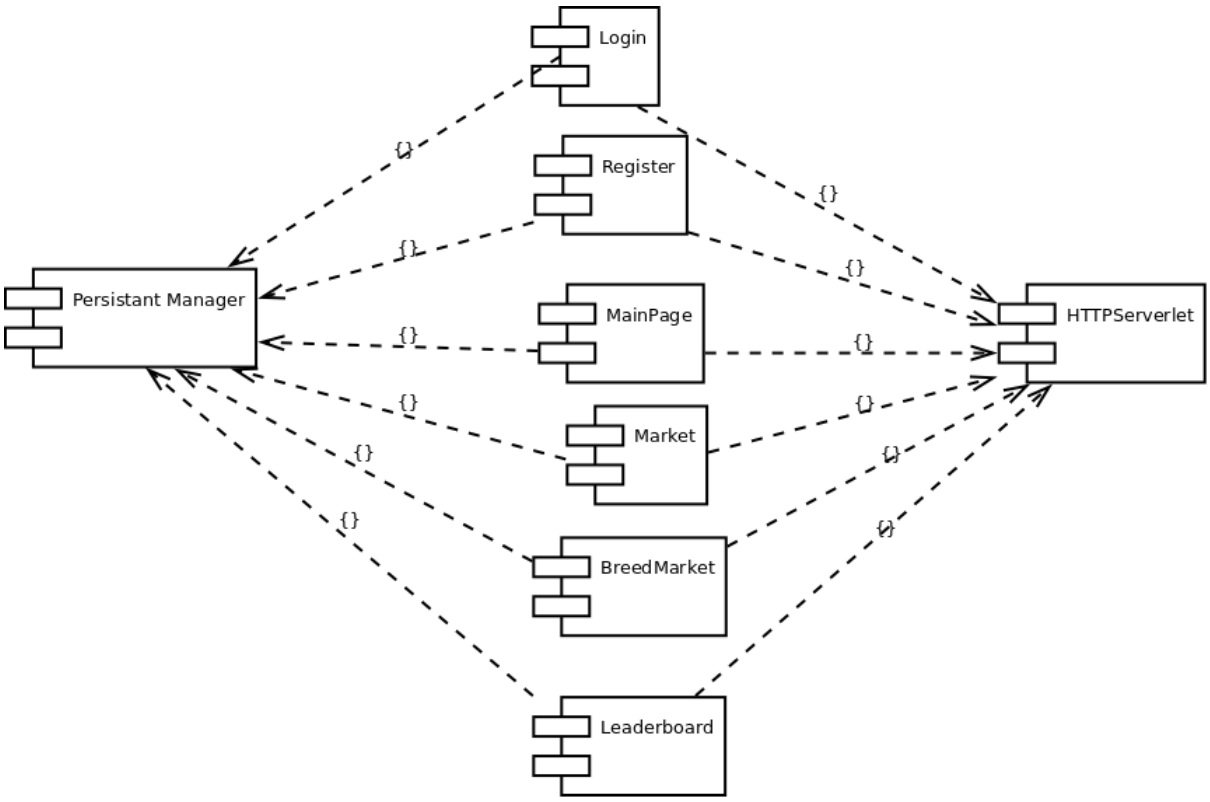
	FightRequest,
FR7	Player, Register, Login, PersistenceManager
FR8	Player, Monster, Notification, Market, FriendRequest, BreedingOffer, FightRequest, MarketOffer, PersistenceManager
FR9	Player, FriendRequest, PersistenceManager
FR10	Notification, Player, Monster(Male), PersistenceManager
FR11	Player, PersistenceManager

3. DEPENDENCY DESCRIPTION

3.1 Component Diagrams

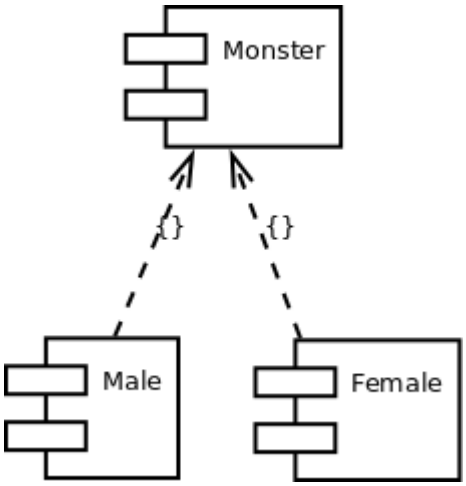
3.1.1 Client

All the pages in the front-end use both the Persistence Manager and the HTTPServerlet. The PersistenceManager provides data for the webpages, and the HTTPServerlet serves the webpages to the web browser. This diagram describes the relationship between the different aspects of our project.



3.1.2 Data

This diagram represents how the classes depend on another within the project.



4. INTERFACE DESCRIPTION

4.1 Overview of Classes

4.1.1 BreedingOffer

This class holds the information concerning a breeding offer including the players and monsters involved, the money required for a player to accept the offer and the date upon which it was placed.

4.1.2 Female M.

Females do not fight, but they do have children and can be offered for breeding, breeding will happen within a female class. After the breeding is done the Female monster and any offspring are returned to the owner of the monster

Female monsters also have battle stats, but they are no more than genetic traits for them, but not for their children as these attributes are passed on to their offspring

4.1.3 Fight Request

This is the class that stores the fight requests given it will store links to the players and monsters involve

4.1.4 Friend Request

This is the class which models the sent requests and will store the players it is between and whether it has been accepted yet. It will also store the date that it was sent.

4.1.5 Male

This extends the Monster class and such has all the genetic attributes associated with that. Male monsters are used for fighting other monsters and so the methods required to determine the victor in a battle will be within this class.

4.1.6 MarketOffer

This class models the current monster sale offers within our server, this will contain the relevant monster and player and also the price of the monster.

4.1.7 Monster

Is an abstract class that holds all monsters stats like: offensive battle stats (strength, speed, accuracy), defensive battle stats (endurance, armor, dodge), personal stats (name, date of birth, age, owner, age rate, fertility). This class will also be responsible for monster ageing which is dependent on the monster age rate that allows some monsters to age slower or the opposite.

4.1.8 Notification

This Class is responsible for notification events. Players will get notifications if someone decides to fight them or buy their monster. There will also be notifications for breeding and adding the player as a friend.

4.1.9 Player

This holds all player's details that player enters at registration: username, password, email. Also it lists their friends and monsters, their money, notifications created by the Notification class. This class will be used to create player's very first initial monster. The monsters will be stored in here as a list of Monster objects.

4.1.10 Breeding Market

This class processes breed requests and actually sends or shows them.

4.1.11 Leader Board

This class models the leaderboard and will store the list of the wealthiest players and the amount of money they have.

4.1.12 Login

This class models the login data of a username/email address and password.

4.1.13 Main Page

This class models the main page shown to the user in their browser.

4.1.14 Market

This class is responsible for displaying the market which is a hash table of monsters for sale and also of those that you put for sale. It also processes buy and sell requests.

4.1.15 Persistence Manager

is a class that manages all persistent data and communicates with database. Its responsibility is object manipulation. It can get friends, monsters, players, notifications; as well as add these things.

4.1.16 Register

This class is for registration it will create a player with their attributes (username, password, email) which will be saved in a player class.

4.2 Class Skeletons

This sections shows some code skeletons automatically generated from our UML designs, these adhere to the Java coding specifications [3].

4.2.1 BreedingOffer

```
import Player;
import Male;

public class BreedingOffer

{
    /** Attributes */
    private int id;
    private Player player;
    private Male monster;
    private int moneyCost;
    private Date offerStartTime;

}
```

4.2.2 Female

```
import Male;
import Monster;

public class Female extends Monster

{
    /** Attributes */
    private final int MAX_CHILDREN;
    /**
* Operation
*
* @param monster
* @return Monster[]
*/
    public Monster[] breeding ( Male monster )
    {

    }

}
```

4.2.3 FightRequest

```
import Player;
import Male;
```

```
public class FightRequest


{
    /** Attributes */
    private int id;
    private Player sender;
    private Player reciver;
    private int moneyOffer;
    private Male monster;
    private Male opponentMonster;
    private Date offerSentTime;
    private String figthKey;

      }
```

4.2.4 FriendRequest

```
import Player;

public class FriendRequest

{
    /** Attributes */
    private int id;
    private Player sender;
    private Player reciver;
    private Date offerSentTime;
    private String localKey;
    private String remoteKey;

}
```

4.2.5 Male

```
import Male;
import Monster;
import BreedingOffer;
import FightRequest;

public class Male extends Monster

{
    /** Attributes */
    private final int MAX_RANGE;
    private boolean injured;
    /**
* Operation
*
* @param opponent
* @return boolean
*/
    public boolean fight ( Male opponent )
```

```
    {

    }

}
```

4.2.6 MarketOffer

```
import Player;
import Monster;

public class MarketOffer

{
    /** Attributes */
    private Player seller;
    private Monster monster;
    private Date offerSentTime;
    private int id;
    private int money;

}
```

4.2.7 Monster

```
import MarketOffer;

public abstract class Monster

{
    /** Attributes */
    protected int id;
    protected String name;
    protected Date dob;
    protected float genetic_strength;
    protected float speed;
    protected float accuracy;
    protected float endurance;
    protected float armor;
    protected float dodge;
    protected float age_rate;
    protected float fertility;
    protected float health;
    protected float strength;

}
```

4.2.8 Notification

```
import Player;

public class Notification

{
```

```
    /** Attributes */
    private int id;
    private String text;
    private Player player;
    private Date timeSent;

    /** Associations */
    private Player unnamed;

}
```

4.2.9 Player

```
import Player;
import Monster;
import FriendRequest;
import FightRequest;
import MarketOffer;
import BreedingOffer;


public class Player

{
    /** Attributes */
    private int id;
    private String password;
    private String email;
    private ArrayList<Player> friends;
    private int money;

    /** Associations */
    private Monster unnamed;

    /**
* Operation
*
* @return
*/
    public createInitialMonster ( )
    {
        //
    }
    /**
* Operation
*
* @param friend
* @return
*/
    public addFriend ( Player friend )
    {

    }
    /**
* Operation
*
* @param friend
* @return
*/
```

```
    public removeFriend ( Player friend )
    {

    }
    /**
* Operation
*
* @param monster
* @return
*/
    public addMonster ( Monster monster )
    {

    }
    /**
* Operation
*
* @param monster
* @return
*/
    public removeMonster ( Monster monster )
    {

    }
    /**
* Operation
*
* @param notefication
* @return
*/
    public addNotification ( Notefication notefication )
    {

    }
    /**
* Operation
*
* @return
*/
    public updateMonsters ( )
    {

    }

}
```

4.2.10 BreedingMarket

```
import HttpServlet;

public class BreedingMarket extends HttpServlet

{

    /**
* Operation
*
* @param request
* @param response
* @return
*/
```

```
    public processBreedRequest ( HttpServletRequest request,
HttpServletResponse response )
    {

    }

}
```

4.2.11 Leaderboard

```
import HttpServlet;

public class Leadboard extends HttpServlet

{

    /**
* Operation
*
* @param request
* @param response
* @return
*/
    public processRequest ( HttpServletRequest request, HttpServletResponse
response )
    {

    }

}
```

4.2.12 Login

```
mport HttpServlet;

public class Login extends HttpServlet

{

    /**
* Operation
*
* @param request
* @param response
* @return
*/
    public processLoginRequest ( HttpServletRequest request,
HttpServletResponse response )
    {

    }
    /**
* Operation
*
* @param request
* @param response
* @return
```

```
*/
    public processLogoutRequest ( HttpServletRequest request,
HttpServletResponse response )
    {

    }

}
```

4.2.13 Mainpage

```
iimport HttpServlet;

public class MainPage extends HttpServlet

{

    /**
* Operation
*
* @param request
* @param response
* @return
*/
    public processFriendRequest ( HttpServletRequest request,
HttpServletResponse response )
    {

    }

}
```

4.2.14 Market

```
import HttpServlet;

public class Market extends HttpServlet

{

    /**
* Operation
*
* @param request
* @param response
* @return
*/
    public processSellRequest ( HttpServletRequest request,
HttpServletResponse response )
    {

    }
    /**
* Operation
*
* @return
*/
    public processBuyRequest ( )
    {
```

```
    }
}
```

4.2.15 Persistencemanager

```
import Register;
import Login;
import Market;
import Leadboard;
import BreedingMarket;
import MainPage;

public class PersistenceManager

{
    /** Attributes */
    private final String dbName;
    private final String dbHost;
    private final String dbPort;
    private final String dbUsername;
    private final String dbPassword;
    private Connection connection;
    private String error;

    /**
* Operation
*
* @param query
* @return boolean
*/
    public boolean query ( String query )
    {

    }
    /**
* Operation
*
* @param query
* @return String[]
*/
    public String[] select ( String query )
    {

    }
    /**
* Operation
*
* @return String
*/
    public String getError ( )
    {

    }

}
```

4.2.16 Register

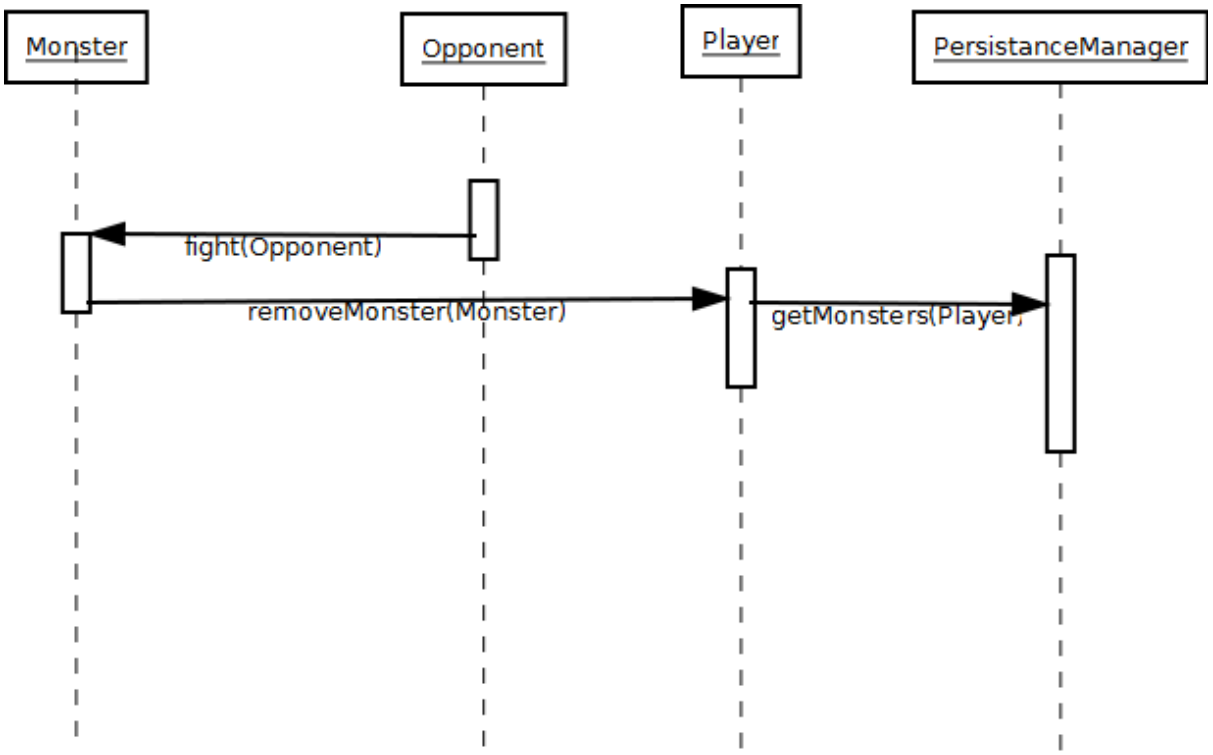
```
import Player;
import HttpServlet;

public class Register extends HttpServlet

{

    /**
* Operation
*
* @param username
* @param email
* @param password
* @return boolean
*/
    public boolean createPlayer ( String username, String email, String
password )
    {

    }
    /**
* Operation
*
* @param player
* @return
*/
    public unregister ( Player player )
    {

    }
```

5. DETAILED DESIGNS

5.1 Sequence Diagrams

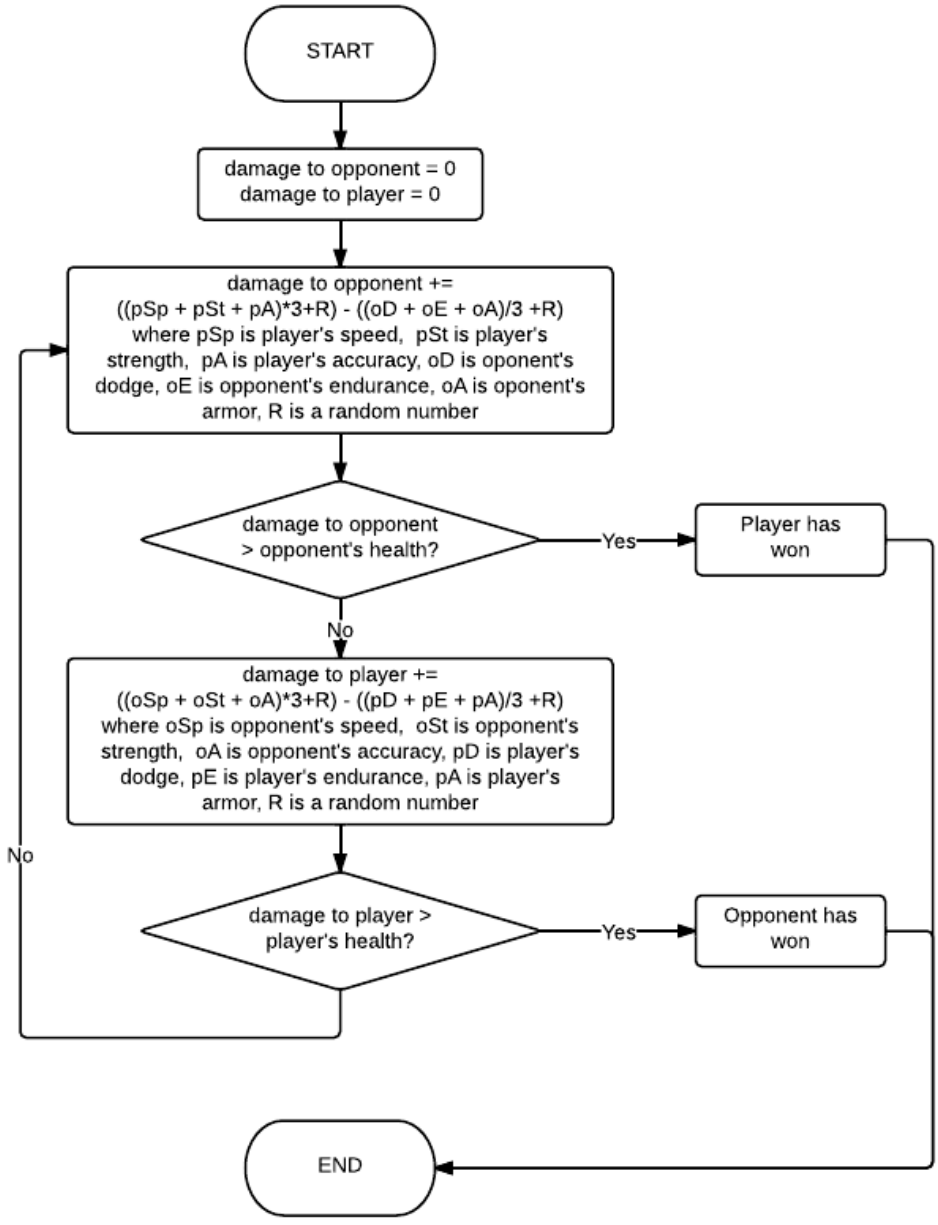
5.1.1 Fighting

This shows how the fight would take place at a slightly higher level than the actual algorithm. The dead monster would be passed back to the player (owner) who would send it off to get removed from the server.



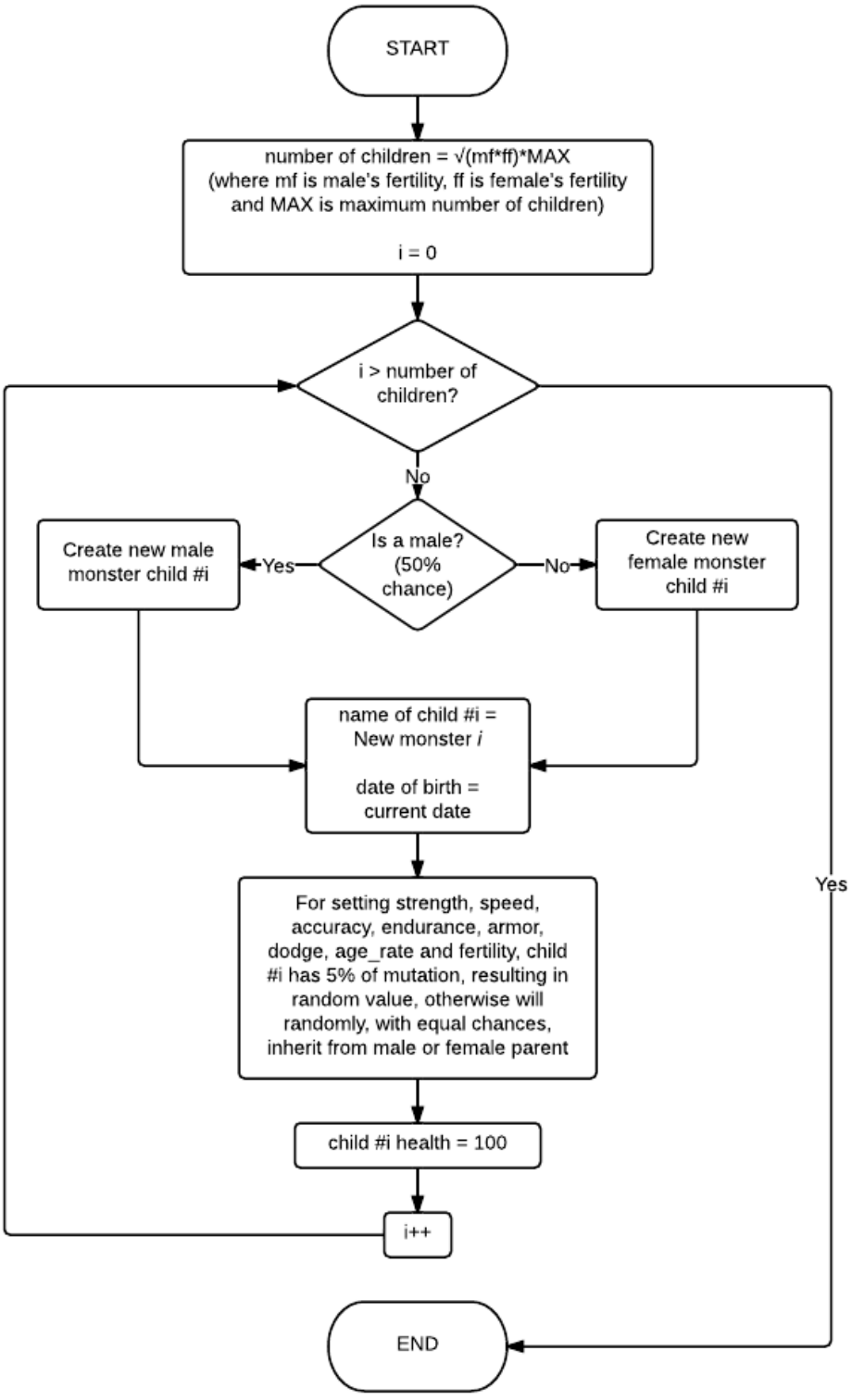
5.2 Significant Algorithms

5.2.1 Battling



```
int MAX_RANGE = 30;

public boolean fight(Male opponent)
{
	float damageToOpponent = 0;
	float damageToPlayer = 0;

	do
	{
		damageToOpponent = (((this.speed + this.strength + this.accuracy)*3)
			-MAX_RANGE + random(MAX_RANGE*2)) -
				(((opponent.dodge + opponent.endurance + opponent.armor)/3)
					-MAX_RANGE + random(MAX_RANGE*2));

		if(damageToOpponent > oppent.health)
			return true;

		damageToPlayer = (((opponent.speed + opponent.strength + opponent.accuracy)*3)
			-MAX_RANGE + random(MAX_RANGE*2)) -
				(((this.dodge + this.endurance + this.armor)/3)
					-MAX_RANGE + random(MAX_RANGE*2));

		if(damageToPlayer > player.health)
			return false;
	} while(true);
}
```

5.2.2 Breeding

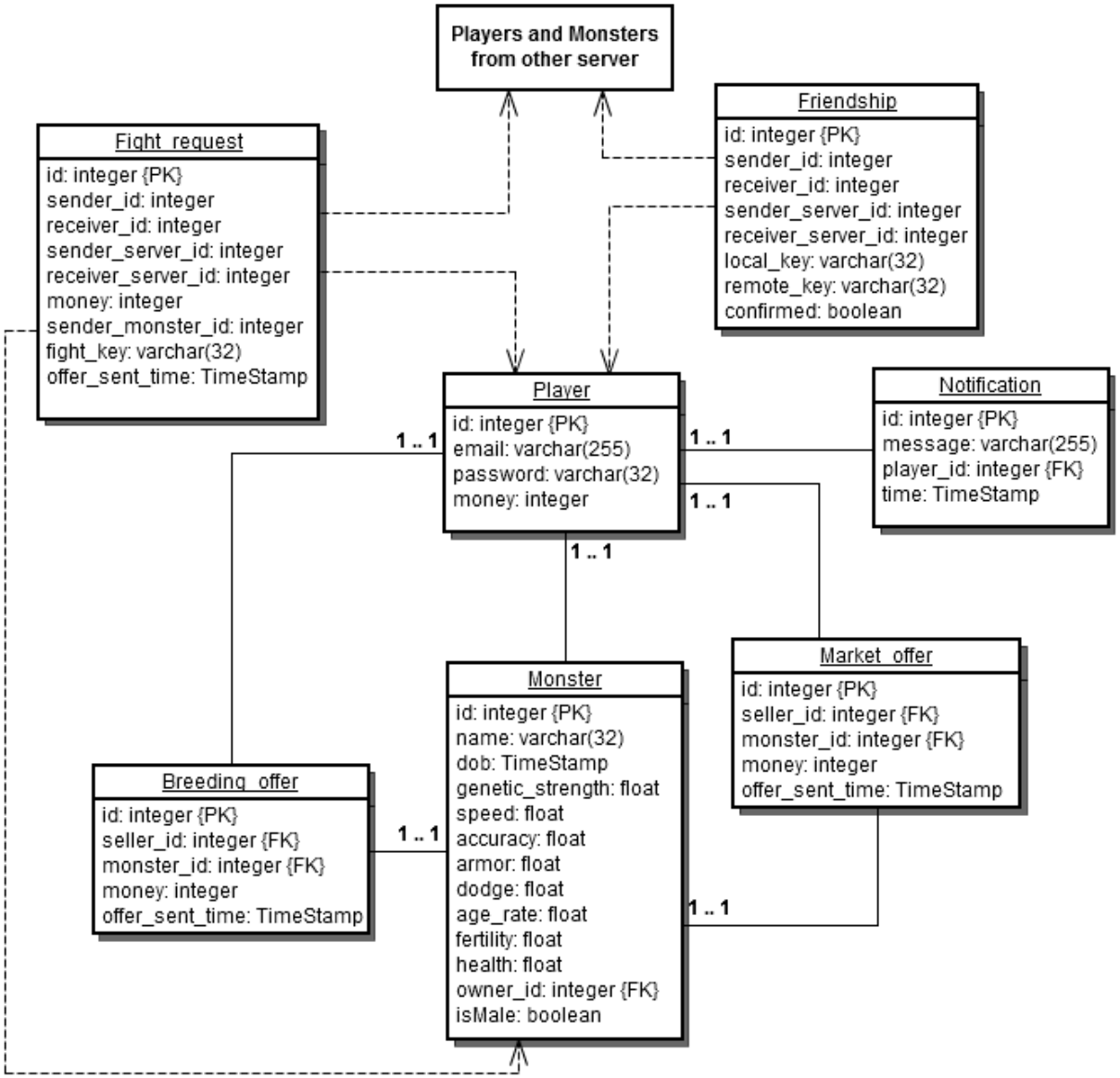


```
public Monster[] breeding(Male monster)
{
        Random random = new Random();

        int numberOfChildren = Math.sqrt(fertility * monster.fertility) * MAX_CHILDREN;
        Monster children[numberOfChildren + 1];

        for(int i = 0; i < numberOfChildren; i++)
        {
                boolean isMale = random.nextBoolean();
                if(isMale)
                        children[i] = new Male();
                else
                        children[i] = new Female();

                children[i].name = "New monster " + i;
                children[i].dob = new Date();

                if(random.nextInt(100)<5)
                        children[i].strength = random.nextFloat(1);
                else if(random.nextInt(100)<50)
                        children[i].strength = strength;
                else
                        children[i].strength = monster.strength;

                if(random.nextInt(100)<5)
                        children[i].speed = random.nextFloat(1);
                else if(random.nextInt(100)<50)
                        children[i].speed = speed;
                else
                        children[i].speed = monster.speed;

                if(random.nextInt(100)<5)
                        children[i].accuracy = random.nextFloat(1);
                else if(random.nextInt(100)<50)
                        children[i].accuracy = accuracy;
                else
                        children[i].accuracy = monster.accuracy;

                if(random.nextInt(100)<5)
                        children[i].endurance = random.nextFloat(1);
                else if(random.nextInt(100)<50)
                        children[i].endurance = endurance;
                else
                        children[i].endurance = monster.endurance;

                if(random.nextInt(100)<5)
                        children[i].armor = random.nextFloat(1);
                else if(random.nextInt(100)<50)
                        children[i].armor = armor;
                else
                        children[i].armor = monster.armor;

                if(random.nextInt(100)<5)
                        children[i].dodge = random.nextFloat(1);
                else if(random.nextInt(100)<50)
                        children[i].dodge = dodge;
                else
                        children[i].dodge = monster.dodge;

                if(random.nextInt(100)<5)
                        children[i].age_rate = random.nextFloat(1);
                else if(random.nextInt(100)<50)
                        children[i].age_rate = age_rate;
                else
                        children[i].age_rate = monster.age_rate;

                if(random.nextInt(100)<5)
                        children[i].fertility = random.nextFloat(1);
                else if(random.nextInt(100)<50)
                        children[i].fertility = fertility;
                else
                        children[i].fertility = monster.fertility;

                children[i].health = 100;
        }
        return children; }
```

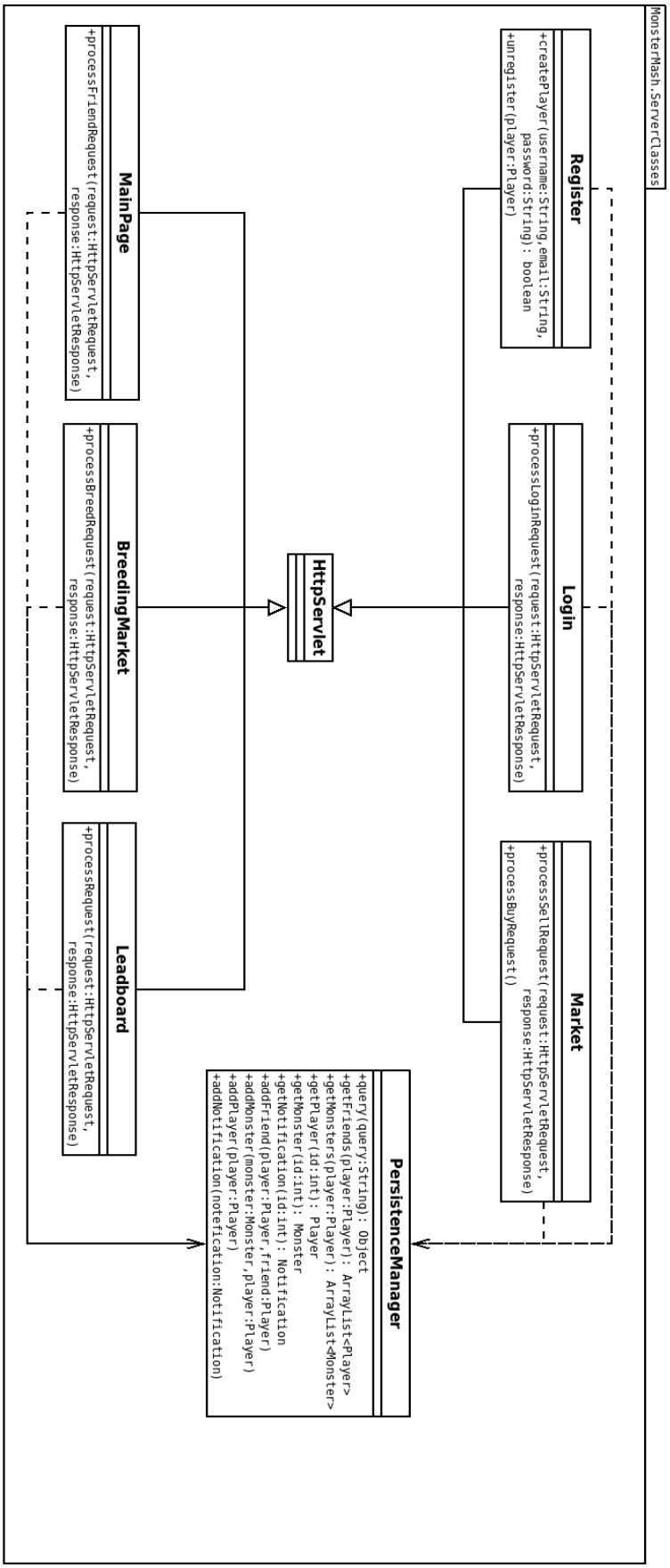
5.3 Significant Data structures

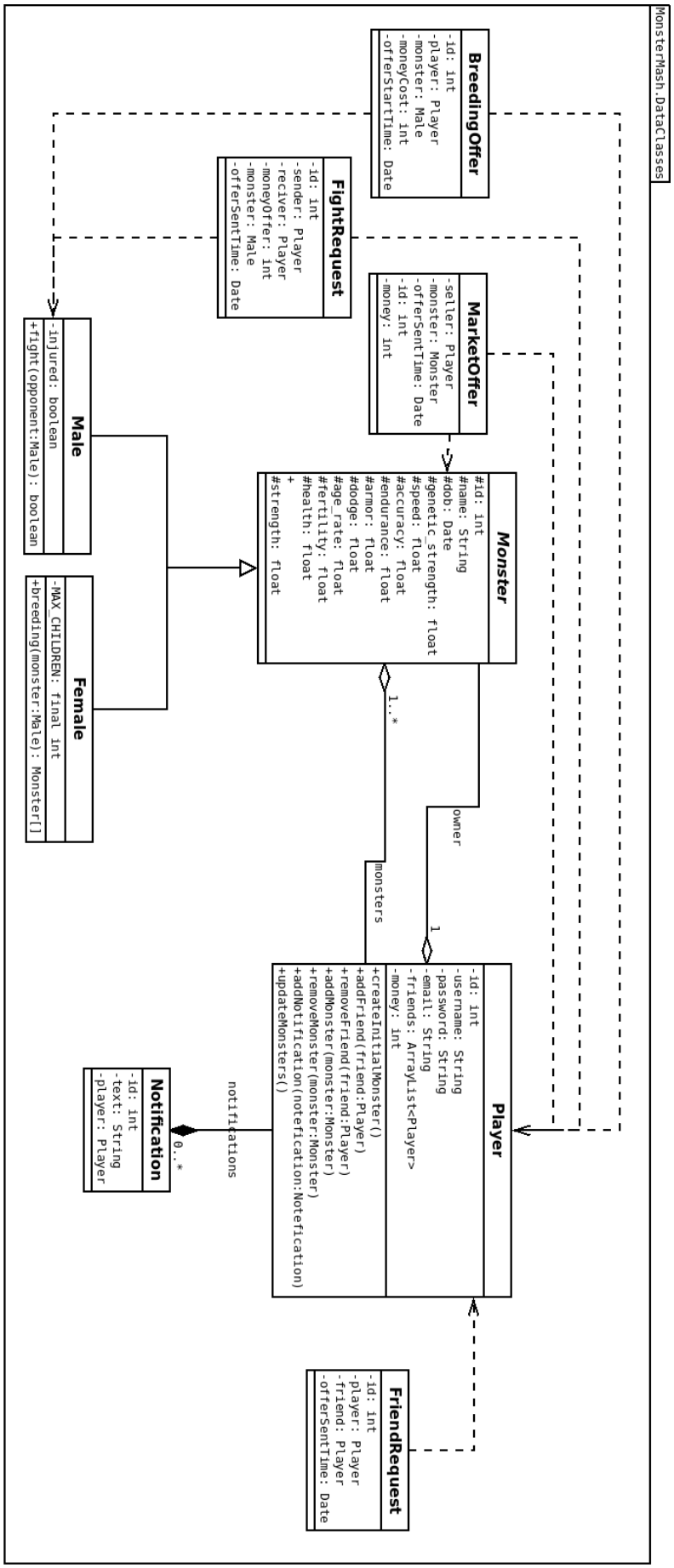
5.3.1 Database Description

The database is used to store all data, so that data is not lost when the server is down. The instance variables of the java objects are saved so that they can re-instantiated. In the diagram below the *fight request* and *Friendship* tables do not have ForeignKeys for monster and player because not all players and monsters are stored in our database.



5.4 UML Designs




MonsterMash.DataClasses
BreedingOffer
-id: int
-player: Player
-monster: Male
-moneyCost: int
-offerStartTime: Date
FightRequest
-id: int
-sender: Player
-reciver: Player
-moneyOffer: int
-monster: Male
-offerSentTime: Date
MarketOffer
-seller: Player
-monster: Monster
-offerSentTime: Date
-id: int
-money: int
Male
-injured: boolean
+fight(opponent:Male): boolean
Female
-MAX_CHILDREN: final int
+breeding(monster:Male): Monster[]
Monster
#id: int
#name: String
#dob: Date
#genetic_strength: float
#speed: float
#accuracy: float
#endurance: float
#armor: float
#dodge: float
#age_rate: float
#fertility: float
#health: float
+
#strength: float
1..*
monsters
owner
1
Player
-id: int
-username: String
-password: String
-email: String
-friends: ArrayList<Player>
-money: int
+createInitialMonster()
+addFriend(friend:Player)
+removeFriend(friend:Player)
+addMonster(monster:Monster)
+removeMonster(monster:Monster)
+addNotification(notefication:Notefication)
+updateMonsters()
notifications
0..*
Notification
-id: int
-text: String
-player: Player
FriendRequest
-id: int
-player: Player
-friend: Player
-offerSentTime: Date

REFERENCES

[1] **Software Engineering Group Projects Monster Mash Game Requirements Specification**
Config Ref: SE.CS.RS

[2] **Software Engineering Group Projects Design Specification Standards**
Config Ref: SE.QA.05A

[3] **Software Engineering Group Projects Java Coding Standards**
Config Ref**:** SE.QA.09

DOCUMENT HISTORY

Version	*CCF No.*	*Date*	*Changes made to document*	*Changed by*
1.0	N/A	05/12/12	Creation of document, all major elements added	G12
1.1	N/A	06/12/12	Overhaul of many documentation and format elements	jau1