Software Engineering Group Projects
Monster Mash Game
Requirements Specification

Author:	B. P . Tiddeman
Config Ref:	SE.CS.RS
Date:	7th October 2012
Version:	1.1
Status:	Second draft

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this document

This document describes the requirements for the CS22120 Software Engineering Group Project 2012 for most students studying Computing (including BIT and IC students). It should be read in the context of the Group Project, taking into account the details of the group project assignment and the group project Quality Assurance (QA) Plan [1].

1.2 Scope of this document

This requirements specification describes the functions of a "monster" breeding, training and competing computer game, and the attributes that are expected from the finished product. It also describes the requirements for the process of constructing the system.

1.3 Objectives

The objectives of this document are:

- To describe the background to the CS22120 group project application for 2012 (Monster Mash Game)
- To provide details of the criteria that the group project product must meet
- To describe the types of interaction with the system which must be supported

2.GENERAL DESCRIPTION

2.1Product Perspective

This project will produce a monster web-based game. The system client will interact with a server, which will also be developed as part of the project. Each player client hosts a "monster farm" which contains a collection of monsters (initially just one). Monsters have a set of "genetic" attributes (colour, size, aggression, sex etc) stored as a string of bits, which are swapped when breeding. They also have some other attributes such as age and health. Players interact with other players to breed (for a fee), to fight monsters (for prize money) and to purchase monsters, etc.. Monsters can die in fight contests, from old age or disease. The aim of the game is to make as much (virtual) money as possible..

The game play screen should show the user's list of monsters within it, including the monster's attributes. The user should see a list of their "friends" with which they can interact. They should have the option to send friend requests (via unique identifier such as email address), confirm or deny friend requests they receive. For existing friends the options are to request a prize fight, to offer or request monster mating, to offer monsters for sale or purchase monsters off other users.

An important component of the system is server-server interaction. Different user's will have their accounts hosted on different servers. The servers need to communicate to manage friend requests, matches, purchases etc between users hosted on different servers. The protocols for communicating between servers will need to be agreed between groups, so that servers implemented by the different groups can work together.

The product will run on standard web browsers e.g. those installed on the IS desktop machines within the Department of Computer Science, and the user will be able to use a web-based interface to play the game. The server side component will be used to store details of users, monsters, friends etc..

2.2 Product Functions

The product will provide the following features:

- A server program with supporting storage and communication interface (protocols etc) for storing details of users and their monsters and communicating these to the clients (browsers).
- The server should also maintain and manage a friends list for each registered user and allow new users to register.
- The system should provide a method for connecting with friends with which to play the game. These may be hosted on a different server.
- A web-based user interface with which to play the game including the ability:
    - To offer male monsters for breeding (for a fee).
    - To send fight requests to friends (a "Monster Mash").
- Fights are decided based on the characteristics of the monsters with a random element. The prize is based on winner takes all and losers are killed.

Animated elements, such as visualisation of the fights is not required, but the software should be developed in such a way that these elements could be added later.

2.3 User Characteristics

The software is intended as an educational tool, designed as part of a programme to popularise and inform the general public about aspects of evolution and genetics.

The game players will be young people who play web-based games and are (or could be made) interested in evolution. They will expect the kinds of features and interfaces that are normally seen for such games.

3.SPECIFIC REQUIREMENTS

3.1 Functional Requirements

FR1 Server-based authentication
The server will be used to authenticate a user, allowing them to log-in or register from their browser.

FR2 Server friends list
The server will maintain a list of friends for each user. User's will only be able to interact directly with their friends. Friends will be identified by their email address and added by a request-confirm mechanism.

FR3 Server monster list
The server will maintain a list of the monsters owned by each player and their attributes. These include genetic attributes and phenotypic attributes (such as age, health etc). The server will manage the monster lifecycle i.e. mating, birth, ageing, illness, injury and death. New users should be allocate a basic (random) monster and a small pot of virtual money.

FR4 Server monster mash management
The server will handle "monster mash" with a (virtual) cash prize available. The system will assign a prize value to each monster based on their characteristics. Users can select one of their monsters and challenge one of their friend's monsters to a match. The friend can accept or decline the challenge. If they accept, the server will decide the winner based on the characteristics of the monsters along with an element of random chance (see Appendix A for an outline suggested algorithm). The server "pays" the winner the prize value of the loser.

FR5 Server-server communication
The server should be able to communicate with other servers using a standard protocol (agreed between groups) in order to play the game (add friends, buy/sell monsters, arrange monster breeding, manage fights, etc).

FR6 Client options
The client will allow users to interact with the system i.e. register/unregister, add/remove friends, offer for sale/ buy monsters, offer for breeding / purchase breeding, etc.

FR7 Startup of software in browser
When the software first starts, it will display a set of choices for the user as follows:

- Log in
- Create new account

Once logged in the system should provide an option to log-out. This will take the user back to the initial log-in/ register screen.

FR8 Game display in browser
When the player has logged in they should be able to see a list of their monsters (with status info), their friends (with offers of monsters for sale and for breeding), challenge requests (with prize money etc) and have options to interact with these options as described in FR6.

FR9 Friend matching
The system should allow users to send a friend request to other users of the system (identified e.g. by their email) and to accept or reject requests sent to them. On accept the friend would be added to the friend list.

FR10 Fight notifications
Following a fight that the user has entered, the monster lists off all competitors should be updated. Loser's monsters should be removed from their list, the winner will have the prize money added to his account and the monster's status will be updated (accounting for injuries etc).

FR11 Friends rich list
A user should be able to see a list of his friends (including himself) and the wealth of each, ordered by wealth.

3.2 External Interface Requirements

EIR1 Appearance of Interface

The program should conform to usual look and feel guidelines for web-based applications.

3.3 Performance Requirements

PR1 Response of program to user input

The user should feel like the system is responding to them at all times during game play. There should not be any perceptible lag between attempting a game action and the system responding.

PR2 Target computer for system

The client software produced should run correctly on standard browsers (i.e. one of the browsers installed on the IS desktop). The servers should also run either on the Department's or University's systems or a third-party system, but should be accessible from the department for testing.

3.4 Design Constraints

DC1 Use of Java

It is corporate policy to use Java for the server side development. The precise choice of platform is left to you, e.g. it could be running Java servlets in Glassfish or Tomcat, or hosted on Google App Engine (in Java).
The chosen platform should conform to industry standards and be justified in terms of availability, robustness, scalability and maintenance costs.

DC2 Reuse of existing software

Use of existing classes for basic data structures is encouraged, as is full use of the standard class libraries, but the main components of the system should be the team's own code. At the end of the project, you will consider whether the project is good enough to release as open source. This means that you must not use any licensed material such as code, pictures or development code that might preclude that possibility. Use of any other third-party code should be acknowledged, and if there is any doubt that it's use is acceptable it should be discussed with the group tutor.

3.5 Other Requirements

The project will be developed in line with the group project QA plan, detailed in [1]. Relevant coding standards and configuration procedures will be decided depending on the tools chosen to carry out the project.

REFERENCES

[1] QA Document SE.QA.01 - Quality Assurance Plan.

Appendix A - Monster lifecycle

A monster's genetic characteristics influence its ability to fight, breed etc. These characteristics interact with environment and age over the monster's lifetime. All monsters should be defined by a string of bits of a fixed length, defining the genome. When breeding each value in the new genome is selected at random from one of the two parent genomes i.e.

```
child_value = r>0.5 ? dad_value : mum_value;
```

where r is a random variable between 0 and 1.
There should also be a small probability of a random mutation to any value when breeding e.g. for a 5% probability of mutation

```
child_value = r>0.05 ? child_value : random_value;
```

again r is a (different) random value between 0 and 1 and random_value is a new random value in the desired range for the attribute.

The number of offspring from a single mating should be governed by a combination of the two parents' fertility. For example if one parent has fertility f_1 and the other f_2 (both in the range 0 to 1) a reasonable value might be:

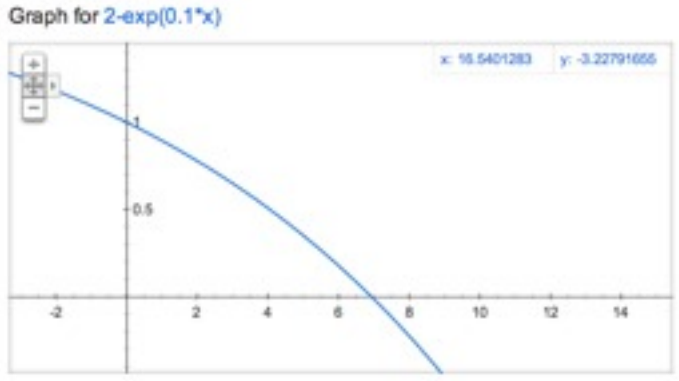
```
number_of_children = Math.sqrt(f1*f2)*max_number_of_children;
```

The offspring will be added to the female's owner's monster collection.

The monster's health will decline over its lifetime e.g. as

```
base_health = 2-Math.exp( age*age_rate);
```

where the age_rate is a genetic characteristic. This function looks like this (with age_rate=0.1):



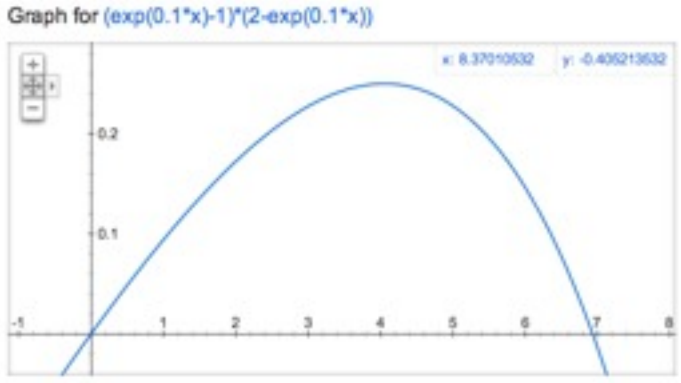
So health decreases from 1 to zero over the lifespan (of about 7, in whatever units you select). The base_health can be reduced by injury i.e.:

```
health = base_health - injuries;
```

when health becomes less than or equal to zero the monster dies. Strength could follow a similar pattern, but rather than declining over the lifetime could rise as the monster grows and decline as it ages e.g.

```
strength = genetic_strength*(Math.exp( age*age_rate)-1)*(2-Math.exp(age*age_rate);
```

This looks like (for age_rate=0.1, genetic_strength=1):



So strength starts small (child monsters), peaks sometime during the lifetime and then goes into decline.Fights should be managed using a combination of aggression, strength and defence to select the winner. The fights could be implemented as a series of random probabilities (rather like repeated throw of a dice). For example (in *untested* pseudo Java code), assuming an array of monsters in the fight (usually of initial length 2):

```
update(monsters); //make sure health, strength etc is up-to-date
while (monsters.length>1) {
    int i = randomInt(monsters.length); //choose a random monster, or could loop over all
```

```
        double r = random();
        if (r<monsters[i].aggression) { // decide if monster i attacks this go
            // Monster i attacks
            int j = randomInt(monsters.length-1);
            if (j>=i) j++; //monster doesn’t attack itself
            // now j is monster to attack
            float damageToI = monsters[j].aggression * monsters[j].strength
                                * (1-monsters[i].defence) * random();
            float damageToJ = monsters[i].aggression * monsters[i].strength
                                * (1-monsters[j].defence) * random();
            monters[i].injuries+=damageToI;
            monters[j].injuries+=damageToJ;
            if (monsters[i].injuries>monsters[i].base_health) remove(monsters, i);
            if (monsters[j].injuries>monsters[j].base_health) remove(monsters, j);
        }
    }
```

DOCUMENT HISTORY

Version	*CCF No.*	*Date*	*Changes made to document*	*Changed by*
1.0	N/A	25/09/12	First Draft	BPT