Software Engineering Group Project
Final Report

Author:	Joshua Bird, Phil Wilkinson, Tom Hull, Dave Haenze, Chris Morgan, Kamarus Alimin, Szymon Stec, Lewis Waldron
Config Ref:	SE_02_FR_01
Date:	15-02-2013
Version:	1.1
Status:	Final

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this Document

The purpose of this document is to give a summary of the complete project in terms of a final report.

1.2 Scope

This document should be written by the relevant person(s) (mainly by the project leader) to give aspects of the final result of the project in detail. This should be read and agreed upon by all project members where there is objectivity.

1.3 Objectives

The aim of this document is to cover the following aspects of the project in the format of a final report:

1. The End-of-Project Report.
2. The Project Test Report.
3. The Project Maintenance Manual.

It also gives details of other additional evaluations for the project, and whether the result is successful or not.

2. END OF PROJECT REPORT

2.1 Management Summary

During the production of this project, many of the required functionality has been achieved, these will be listed below and how well a state they do work in. The following contains what parts of the system are not functioning properly.

Using the use case diagram drawn in the Design Documents, the state of the requirements are as follows:

Working - Buying/Selling Monsters, Breeding Monsters (With no validation however), Challenging Monsters, Accepting Challenges, View Friend Wealth, Sending Friend Requests, Accepting/Declining Friend Requests, Registering, Logging In, Logging Out, Display Monsters and Friends and Deciding Monster Fight outcomes.

Not Included - Deleting Friends and Un-Registering.

There were a few of difficulties while producing this project although these were mitigated as much as possible. The major problem we encountered was allocation of roles in the group were organised, in the first meeting held positions were given to members of the group, but not all of the group members arrived and as such we were not aware of everyone's strengths and weaknesses. As a result certain roles were not ideal for the group, for example, Tom Hull was given ‘Dept. Leader’ and as it turns out ended up as a lead programmer. Dave Haenze was given the role of Dept. QA Manager, and was also a lead programmer.
After the first meeting it was apparent that a few of the unassigned members were not on ‘programming’ modules.
In order to mitigate the problem, the members of the group who were not on heavy programming modules were given more work to do regarding documentation side of the project, and those who were experienced with web programming were assigned to create the user interface side of the product.
All in all, the team pulled together in this aspect, and the work was spread as equally between everyone as well as possible.

2.2 Historical Account of the Project

In order to complete the project in time, a time-line of deadlines was created in the form of a Gantt Chart, which was included in the Project Plan.

There were a number of key aspects of the projects that needed to be completed and as such they were split apart, each being allocated an amount of time in which that piece of work needed to be completed in. The leader set deadlines for the group that were a bit before each of these deadlines, in order to try and reduce slippage.

The first main event was the Project Plan - this also gave a foot-holding for the rest of the project in itself and had information such as use-case diagrams, in order to try and gain an understanding of what functionality the project had to actually include. This was completed first as it was needed for the rest of the project.

Next was getting a clear understanding of the tasks needed throughout the project. This was a more detailed version of the functionality, and in order to complete this the QA documents had to be read carefully to extract the useful information that was needed to get to grip with the full requirements of the system. A set of tests were then decided for each functional requirement, so that the coders were aware ahead of time what they would need to be adding to the system.

In parallel, the Design of the project was also started, this was because it was decided that the design of the system was just as important as its functionality. The first designs included paper drawn versions that the whole team could look at to understand how the final solution should look.

Classes were designed just around this time too, because it gave the coders a better idea of how the system would be created behind the scenes.

Database design was next: - tables were drawn on paper and fields discussed by the team, to incorporate everyone's ideas about how the system should operate, and what tables were needed ahead of time with what values included in them. More time was given for this than the class diagram.

Across the whole of this, the ‘Prototype Demo Software’ was allocated. The reason for this: we were expected to present a basic version of the system by the end of the first term. To try and get a head start, it was planned to have spike work being collected (for example on servlets and databases) before trying to put together a functioning prototype.

The server functionality for the prototype was split into sections, to try and reflect this idea of spike work being collected over a period of time, with the work stretching beyond the end of the first term for more difficult work like ‘Monster Lists’. This kept the initial required sections of work like the communication between the server and databases to the front of the required work to be done.

Then lastly were plotted the deadlines for the submission of work and testing and integration week. These were there as the final milestones needed for the hand in of all of the work, and an indication of where the compiling of spike work would be done and the producing of the final version of the program and documentation.

In order for the team to work effectively on all of the work needed, this plan covered the entire span of the projects lifetime, in order to provide detailed information on how far the project was currently at. If at a particular date the project was not as far as the plan indicated, it would tell the team that the workload needed to be increased in order to maintain the quality of the final product.

2.3 Final State of the Project

In the final project, the majority of the functionality is working, except for server to server between groups however. What is working and what isn’t will be discussed below in more detail the management summary:

Firstly, we had issues with the database working near the end of the project, and as such, some functions worked, but in order for the servlets to display what was in the database, the user would have to logout, and back in to see the updates.

Buying/Selling Monsters - This functionality is working, users are limited to buying and selling monsters between their friends only as required as well as limiting the buying and selling to be within valid prices. This only works between users in this group, inter server communication does not work. The user would have to logout and log back in to see the updated monster list.

Breeding Monsters - This functionality does work, users are able to breed with their friends monsters (if they are up for breeding). However the validation that is present for buying/selling is not working for breeding, so unfortunately wrong values are able to be entered, like minus values. Needs the user to logout and login to update.

Challenging Monsters - The ability to Challenge Monsters does work, and as required only between friends. The user selects a monster they wish to fight, and then a monster of their own on the next screen to send out the challenge. Sometimes viewing the challenges sent doesn't always show up your challenges you have pending.

Accepting/Declining Challenges - The ability to accept or decline challenges does work in the system. By selecting accept challenge the fight will start and a winner will be selected. The loser's monster will be killed and removed from their monster list. If the user selects decline challenge, the fight will not take place. However the challenge is not removed from the list which it should do.
Another issue with the requests is that if the monster dies due to a fight or "natural causes" the request will not be removed. Allowing in some cases for a fight to still take place, even though the monster is already dead.

View Friend Wealth - This functionality works, however it is only displayed within the friends list itself, not on a separate page. So when the user looks at their list of friends, at the bottom of the page their friends are listed in order of how much money they have.

Sending Friend Requests - This functionality works, upon typing in a valid users email into the add friend box and clicking enter the request will be sent. The user can display all of their outgoing requests. Validation is also in place to prevent the user from adding an invalid email and also one that has already been added as a friend.

Accepting/Declining Friend requests - The user is able to accept and decline incoming invites from another player. Upon accepting, the friend list will be updated (after logging out and back in again). Upon selecting decline, the friend request is removed.

Registering - Registering works as intended, the user enters their desired username (email) and password, both which are validated to check they are valid. Their user is then created and given some monsters.

Logging In - Works, upon logging in the user is taken to their list of monsters, where they can browse to other pages or check the stats of the monsters.

Logging Out - The user is able to log out and end the session, which will allow another user to log in on the same device.

However there are some problems with the project:

Un-Registering - Unfortunately Un-Registering was not implemented.

Deleting friends - This function was not added to the project.

There were some UI problems, which meant that some of the information for monsters and such overlapped buttons and other text boxes on the screen. This was due to some errors in the CSS with tables mostly.

Risk assessment - After receiving feedback on the risk assessment we found that it is too long and the risks should have been grouped together into categories.

2.4 Performance of Each Team Member

2.4.1 Phil Wilkinson (Agreed upon):

Phil was designated the role of QA Manager, throughout the project it was his task to look through any documents/work that had been created and check to make sure that it was completed to an acceptable standard. He then had to make sure that all work was maintained to match the QA specification.

Throughout the project I would give Phil documents from other members/or tell him where they were located in our Git repository, so that he could manage them.

He also took it upon himself to restructure the documents when looking at them so that they fit better to the pages and looked nicer which I was very impressed with.
In integration and testing week Phil also managed the Git repositories and separated the different parts of work into different branches to allow finding work easier.

On the last day Phil also had to take on a role of another member and redo the user interface of the project as changes needed making after coding had been done.

Phil's performance on a whole was outstanding and I was never let down when asking him to do anything with the project.

2.4.2 Kamarus Alimin (Agreed upon):

Kamarus was designated the role of lead tester, he was to work out what tests needed to be performed and to document anything to do with testing.

At the start of the project Kamarus informed me that he was not comfortable coding anything, and requested to be given the role for testing, to which I agreed to as we had a lot of group members working on documentation.

Kamarus was also given the task of completing a risk assessment for the project, when checking through it, it was quite a bit too excessive, which wasn't too bad. It did need some editing for grammar and spelling however. After a few changes the risk assessment took shape though.

Kamarus worked with Szymon on the test table, each completing half of it. And also created the Final Test Report. Both were completed, and after grammar/spelling checks from Phil with some reformatting were usable.

Overall Kamarus completed the work set, be it a bit late at times, and I believe he completed them to the best of his ability.

2.4.3 Dave Haenze (Agreed upon):

Dave was assigned the role of deputy QA manager in the first meeting, however it became apparent very quickly that he would be more useful as a lead coder. As such he did not perform much document quality assurance or writing many documents.

For the main part of the beginning of the project Dave didn't have much to do, except discussion of class diagrams and such, but was given spike work into researching what repository system to use. Under his advice we went with Git, we had a few issues with it, but in the end Dave managed to work out how to solve these issues.

Dave worked on creating algorithms for the Monster's functions in the system, a lot of these were completed in the first term, however due to problems with Git a lot of the code was lost. Dave recovered in the integration week and created more elaborate algorithms for the monsters than before.

In the last week Dave worked closely with Thomas to finish a lot of the code and at the end java doc's. He also added various bits of validation throughout the system.

Overall Dave worked very well and completed all of the work assigned to an excellent standard, and also went beyond and helped with the final report in places.

2.4.4 Szymon Stec (Agreed upon):

Szymon was assigned the role of a programmer to start, as he was one of the few not given a role that was on a programming degree scheme. This meant that he worked along side Dave and Thomas for the most part.

I also set Szymon to complete some documentation such as the class diagram and half of the testing table. This was because he was not set as much spike work as the others.

During implementation and testing week Szymon worked heavily on the Friends side of the code, a lot of problems were encountered here due to how the database was structured, in the end however the problem was resolved.

Towards the end of the week Szymon wrote the majority of the JUnit tests in the project and helped to complete the Java Documentation as well.

All in all I believe that Szymon worked hard and completed the tasks set to the best of his ability. Any work set was given on time and complete to an acceptable standard.

2.4.5 Thomas Hull (Agreed upon):

In the first meeting Thomas was assigned the role of deputy project leader. During the project I don't believe there was a need to take on this role much however.

It became apparent very soon into the project that Thomas would be "Lead programmer" and as such wrote most of code in the system. He was assigned the task of researching servlets soon after starting the project.

Thomas recommended using GlassFish and so we decided to go with that, he read up on it a great deal and completed a lot of spike work with it which we adopted into our own project.

Thomas also researched ObjectDB databases and implemented it with the servlet work that had been created. As a group we decided to use both of these in the end system.

During implementation and testing week Thomas worked heavily on coding with Dave and Szymon, spending most of every day at the university. He worked on all aspects of the project, mostly with database interaction, and even wrote instructions for the others to use it.

Overall Thomas exceeded my expectations and worked very hard throughout the whole of the project, producing consistent work of a high standard.

2.4.6 Lewis Waldron:

Lewis was assigned with creating the user interface for the project. Using HTML and CSS, he asked for this role because he was not confident with coding.

Lewis was asked to create some designs of the pages based on the use case diagrams which were used in the documentation. This work was received and was to an acceptable standard.

Lewis was also tasked with creating basic pages in html to use for the prototype. These pages were created but in the end were not used for the prototype because Thomas had edited them to fit the code written.

Everything in the first semester went well and I received work on time by Lewis, however in the second semester there were some issues.

He wrote a good, solid initial framework for the designs with CSS in place, however as the code was being written it needed to be changed to fit this. In the end Phil Wilkinson had to redo most of it on the last day because we could not contact Lewis. As such me and Lewis do not agree on what is written here.

2.4.7 Christian Morgan (Agreed upon):

Christian was put in charge of writing most of the documentation in the project, this was agreed upon because he did not feel confident with programming. This did mean that Chris had a lot of work to complete in the first semester however.

Chris was put in charge of a lot of docs but some sections were written by others who had more knowledge of those areas (such as class diagrams). When Chris was not sure how to complete a section of the

work he asked for help from those in the group that did, which was very good because it meant that documents were completed in time and I always knew how far the docs were in terms of completion.

In integration and testing week Chris was assigned to go through the documents where we had received feedback on them and to update them accordingly. As such he was not required to be present for much of the week.

Overall Chris worked to a very high standard. All of the documents I received had excellent content, only a few minor grammatical mistakes needed to be edited.

2.4.8 Joshua Bird (Agreed upon by Dave, Phil, Szymon, Chris, Thomas):

In the first meeting I was allocated the role of group leader, this was probably because I had already tried to contact the group to work out who was confident in what aspects of the work before we had even had this meeting.

Throughout the project my role was to manage the whole group and allocate work evenly between everyone, I also had to try and manage time for the whole group.
I did this in a few ways, for example I would set deadlines for work to be handed in earlier than the official deadline, so that we would have extra room to edit the documents and prevent slippage.

In the first semester I had to try and get to grips with the system as a whole as soon as possible, in order to try and work out what major tasks needed to be completed and to get an idea of how I could split the work between the group. I tried to split the work, giving those who were strongest in certain areas the work they would be most comfortable with. I believe that I managed to do this.

In integration and testing week I had to work out what was left to do and try and split it down into tasks to make it easier. I tried to think of milestones that we needed. When we were given the list of tests that we would be checked on I broke it down into bullet points that were easy to read as a "checklist". I also have written a lot of the Final Report, along with Phil and Dave.

2.5 Evaluation of the Team and the Project

I believe in the end the whole group worked well together as a team. We had quite a lot of work to complete between us and had quite a mixture of personalities and members with a variety of different strengths. At the start of the project we probably encountered the majority of the negative sides of this, some members were not confident with strangers and as such did not speak up much in meetings.

By the end I think that this was mostly resolved however, when deadlines came near the group pulled together and finished the work on time. Everyone worked on their own sections of the project which stopped too much clashing of work.

I think that work could have been improved however, there were a few functions in the system that really could have been implemented if management had been better. For example in the last few days the programmers were writing a lot of java doc, and I now realise that this could have been performed by those good at documentation. Which would have given programmers more time to code functions.

I also think that the roles should have been changed after the first meeting, as Thomas and Dave didn't really use their roles that much as they ended up doing a lot of the programming.

Lastly I think that more work should have been required in the first semester, if we had had more working code before integration week we would have had a better standing. This was not entirely our fault though as we did lose a lot of code due to Github behaving strangely.

I think that the most important lesson learnt was the importance of time management, although I feel I did a good job, I did feel that at times when I set a deadline it wasn't taken too seriously and I received work later than I had planned, perhaps I should have been more strict in this regard.

I think that we also learned the value of spike work, without the time invested by Dave and Thomas into servlets/databases/version control we would not have finished the project to the standard that we did.

3. APPENDICES

3.1 Project Test Report

Test Ref	Requirement being test	Pass/Fail Criteria	Reason for failure if any;
SE-N02-001	FR1 + FR6	PASS	
SE-N02-002	FR1 + FR6	PASS	
SE-N02-003	FR1 + FR6	PASS	
SE-N02-004	FR1 + FR6	PASS	
SE-N02-005	FR1 + FR6	PASS	
SE-N02-006	FR1 + FR6	FAIL	Unregister function is unavailable
SE-N02-007	FR6	FAIL	Unregister function is unavailable
SE-N02-008	FR2 + FR6	PASS	
SE-N02-009	FR2 + FR6	PASS	
SE-N02-010	FR2 + FR6	PASS	
SE-N02-011	FR2 + FR6	PASS	
SE-N02-012	FR2 + FR6	PASS	
SE-N02-013	FR2 + FR6	PASS	
SE-N02-014	FR2 + FR6	FAIL	Function to remove friend is unavailable
SE-N02-015	FR2 + FR6	FAIL	“Delete” function is unavailable
SE-N02-016	FR3 + FR6	PASS	
SE-N02-017	FR3	PASS	
SE-N02-018	FR4	PASS	
SE-N02-019	FR4	PASS	

SE-N02-020	FR4	PASS	
SE-N02-021	FR4	FAIL	The challenge function works but the no media were loaded
SE-N02-022	FR6	PASS	
SE-N02-023	FR6	PASS	
SE-N02-024	FR6	PASS	
SE-N02-025	FR6	PASS	
SE-N02-026	FR6	FAIL	Function of buying monster in the market is unavailable
SE-N02-027	FR6	PASS	
SE-N02-028	FR6	PASS	
SE-N02-029	FR6	PASS	
SE-N02-030	FR6	PASS	
SE-N02-031	FR6	FAIL	No error message was shown
SE-N02-032	FR6	FAIL	The Buying function in the monster page is unavailable
SE-N02-033	FR6	PASS	
SE-N02-034	FR1 + FR7	PASS	
SE-N02-035	FR7	PASS	
SE-N02-036	FR7	PASS	
SE-N02-037	FR7	PASS	
SE-N02-038	FR7	PASS	
SE-N02-039	FR7	PASS	
SE-N02-040	FR7	PASS	
SE-N02-041	FR8	PASS	
SE-N02-042	FR8	PASS	
SE-N02-043	FR8	PASS	

4. PROJECT MAINTENANCE MANUAL

4.1 Program description

The program created was a web-based game involving monsters and players. The game involves text-based fights and buying/breeding between users. Users can only interact with friends that they have added via sending requests to each other. The user's aim is to make as much money as they can from their monsters and to be as high on their friend's "rich list" as possible.

4.2 Program structure

Our design uses several different diagrams, primarily a class diagram and sequence diagram. Those can be found in design specification. There is also a Use Case diagram available.
Not counting the JSP files (of which there is one for every servlet), the program consists of several source files. First of all, there are the Java files for the two main objects used by the program, Person and Monster, as well as their respective DAO files (PersonDAO.java and MonsterDAO.java, respectively). There is a .java file for Fight objects, which contain information about the challenger and their monster, among others. Additionally, there are the servlet files, one for each JSP file. These are: FriendsServlet(managing the friends list), MyMonsterServlet(for managing your monsters), MonsterFightServlet(for managing fights and fight requests), LoginServlet(for logging in to the game) and finally Register(creating new users).

4.2.1 PersonDAO.java

public void persist(Person person)
Creates a connection with database server.

public void addFriendRequest(Person person, String email)
Opens connection with database server. Finds person entity in database by unique ID. Adds new friend request to ArrayList of friend requests. Updates database and closes the connection.

public void deleteFriendRequest(Person person, String email)
Opens connection with database server. Finds person entity in database by unique ID. Deletes friend request from ArrayList of friend requests. Updates database and closes the connection.

public void addFriend(Person person, String email)
Opens connection with database server. Finds person entity in database by unique ID. Adds new friend to ArrayList of friends. Updates database and closes the connection.

public void deleteFriend(Person person, String email)
Opens connection with database server. Finds person entity in database by unique ID. Deletes friend from ArrayList of friends. Updates database and closes the connection.

private ArrayList<Person> getAllPeople()
Opens connection with database server. Gets all database entities from the Person table. Adds all entities to the ArrayList. Returns ArrayList.

public ArrayList<Person> getPeoplesArrays(ArrayList<Person> people)
Creates an ArrayList of Persons then returns that ArrayList.

public Person getPersonsArrays(Person p)
Opens connection with database server and gets the Person object by its unique ID from the database. Finally, it returns the Person object.

public boolean lookForEmail(String email)
Uses getAllPeople() method and checks if provided unique email address exists in database. Returns "true" if found, otherwise returns "false".

public Person getPersonByEmail(String email)

Uses getAllPeople() method and searches for provided email address. Returns the Person if found, otherwise returns “null”.

public boolean checkFriendRequestList(Person p, String email)
Opens connection with database server and tries to find the Person entity in the database by its unique ID. Checks if the friends email exists in the person's ArrayList of friend requests. Returns “true” if found, otherwise returns “false”, then closes database the connection.

public ArrayList<Person> getPersonsFriends(Person p)
Returns an ArrayList of the person's friends.

public ArrayList<Person> getPersonsFriendRequests(Person p)
Returns an ArrayList of the person's friend requests.

public ArrayList<Monster> getPersonsMonsters(Person p)
Uses getMonsterByUser() method form MonsterDAO class then returns an ArrayList of the person's monsters.

public void giveFirstMonster(Person p)
Uses generateRandom() method from Monster class and adds the new monster to the new user's monster list.

public Person getPersonByID(Long id)
Uses getAllPeople() method. Finds Person by unique ID.Returns Person if found, otherwise returns “null”.

public void updatePersonsInfo(Person person)
Opens connection with database server and finds the person entity in the database by its unique ID. Updates the person's money then updates the database and closes the connection.

public void updatePersonsInfo(Person person,Fight fight)
Opens connection with database server and finds the person entity in the database by its unique ID. Adds new fight offer then updates the database and closes the connection.

public void deupdatPersonsInfo(Person person,Fight fight)
Opens connection with database server and finds the person entity in the database by its unique ID. Removes fight offer then updates the database and closes the connection.

4.2.2 Person.java

public ArrayList<Fight> getFightChallenges()
Returns an ArrayList of the person's fight challenges.

public ArrayList<Fight> getFightOffers()
Returns an ArrayList of the person's fight offers.

public ArrayList<Fight> getAllFights()
Returns an ArrayList of the person's fights.

public void addFight(Fight fight)
Adds a new fight to the person's ArrayList of fights.

public void removeFightOffer(Fight fight)
Removes a fight offer from the person's ArrayList of fights.

public void removeFightOffer(Person opponent, Monster oppMonster)
Removes a fight offer from the person's ArrayList of fights if fight offer exists.

public Fight getFightByID(Long person, Long monster)
Returns the person's fight if found, otherwise returns “null”.

public ArrayList<String> getActivity()
Returns person's ArrayList of activities.

public void addActivity(String active)
Adds new activity to person's ArrayList of activities.

public ArrayList<String> getAllFriendRequests()
Returns person's ArrayList of friend requests.

public void addFriendRequest(String email)
Adds new friend request to person's ArrayList of friend requests.

public void removeFriendRequest(String email)
Removes friend request from person's ArrayList of friend requests.

public ArrayList<String> getAllFriends()
Returns person's ArrayList of friends.

public void addFriend(String email)
Adds new friend to person's ArrayList of friends.

public void removeFriend(String email)
Removes friend from person's ArrayList of friends.

public void setEmail(String email)
Sets the person's email address.
public void setMoney(int money)
Sets the person's money.

public void setName(String name)
Sets person’s name.

public void setPassword(String password)
Sets the person's user password.

public String getEmail()
Returns persons email address.

public int getMoney()
Returns person's money.

public String getName()
Returns person’s name.

public String getPassword()
Returns the person's password.

public Long getId()
Returns the person's unique ID.

public void setId(Long id)
Sets the person's ID.

4.2.3 Register servlet

All of the methods throws ServletException and IOException, except getServletInfo().

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
Does nothing. Not fully implemented.

protected void doGet(HttpServletRequest request, HttpServletResponse response)
Call the processRequest method.

protected void doPost(HttpServletRequest request, HttpServletResponse response)
Creates a new PersonDAO object for interacting with the database. Assigns request parameters to newly created Strings for checking if an email address is already in the system. If so, displays warning. Otherwise, adds email and password combination to database, then forwards to login page.

public String getServletInfo()
It has no parameters, and simply returns a String “Short description”. Has not been fully implemented.

4.2.4 Login Servlet

All of the methods throws ServletException and IOException, except getServletInfo() and check.

protected void doPost(HttpServletRequest request, HttpServletResponse response)
Sets newly created Strings to request parameters email and password. Checks if user has clicked logout button. If not, creates new session and populates attributes with fields from the database (user's monsters, fight requests, etc.) and forwards to myMonsters.jsp. Otherwise, posts error due to wrong password and/or email being used.

protected void doGet(HttpServletRequest request, HttpServletResponse response)
Does nothing. Not used by the program.

public boolean check(String email, String password)
Creates new PersonDAO for interacting with the database. Returns a 'true' response if email and password combination is correct.

public String getServletInfo()
It has no parameters, and simply returns a String “Short description”. Has not been fully implemented.

4.2.5 MyMonster Servlet

The doGet, doPost and processRequest methods throw ServletException and IOException.

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
Does nothing. Not fully implemented.

protected void doGet(HttpServletRequest request, HttpServletResponse response)
Call the processRequest method.

protected void doPost(HttpServletRequest request, HttpServletResponse response)
Two objects are re-initialized, personDAO and MonsterDAO. The user is retrieved using the respective session attribute. Next, through a series of if statements, it is decided which method to call. Finally, the page is reloaded.

public String getServletInfo()
It has no parameters, and simply returns a String “Short description”. Has not been fully implemented.

public void setBreedOffer(HttpSession session, HttpServletRequest request)
The monsterDAO object is re-initialized and used to find a monster by its ID (provided as a request parameter). A new integer is created and set to the value of the request parameter “breed price”. The Monster object's sale offer is set to this integer, then the database is updated. Finally, the currentAction is set to “changeMonster”.

public void setSaleOffer(HttpSession session, HttpServletRequest request)
The monsterDAO object is re-initialized and used to find a monster by its ID (provided as a request parameter). A new integer is created and set to the value of the request parameter “sale price”. The Monster object's sale offer is set to this integer, then the database is updated. Finally, the currentAction is set to “changeMonster”.

public void setCurrentMonster(HttpSession session, Long id)
A new MonsterDAO object is created and the session attribute "current monster" is set to the Monster that possesses the ID from the method's parameters.

private HttpServletRequest breedMonster(HttpServletRequest request, Person user)
Firstly, the monster IDs are retrieved from the session/request and are used to select the appropriate monsters from the database; these are called "stud" and "bitch". Next the seller is found in the database (the provider of the "stud" monster) and their money increases while the user's decreases. A new Monster object is created, using the breedMonsters method. Its owner is set to the user and then all changes are saved in the database and the request returned.

private HttpServletRequest fightMonster(HttpServletRequest request, Person user)
Firstly, the monster IDs are retrieved from the session/request and are used to select the appropriate monsters from the database. After that, the challenged person is found in the database and a new Fight object is created, using the challenger, the challenged and the two monsters as parameters. Finally, the fight is added to both Person objects and their respective database fields updated and the request is returned.

4.2.6 Friends Servlet

The doGet, doPost and processRequest methods throw ServletException and IOException.

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
Does nothing. Not fully implemented.

protected void doGet(HttpServletRequest request, HttpServletResponse response)
Does nothing. Not fully implemented.

protected void doPost(HttpServletRequest request, HttpServletResponse response):
Method uses user response to execute functions related with persons friends such as:
- sending friend request.
- accepting received friend requests.
- declining received friend requests.
- displaying friends monsters.
- requesting monster fights.
- buying monsters.
- breeding monsters.
- printing friends and friend requests lists.

private boolean checkIfExist(String userEmail):
Method checks if given email exists in the database and returns "true" if found, otherwise "null" is returned.

private HttpServletRequest sendRequest(HttpServletRequest request, Person user):
Method checks if friends email exists in database and if found adds new friend requests to friends ArrayList of friend requests and database is updated, otherwise error message is displayed.

private HttpServletRequest acceptRequest(HttpServletRequest request, Person user):
Method finds Person object form database by persons unique email address, if not found displays en error message, otherwise checks if friend requests exists in person friend requests list, if found a new friend is added to user and request sender friend list. Friend request from users friend requests list is deleted and database is updated.

private HttpServletRequest declineRequest(HttpServletRequest request, Person user):
Method finds Person object form database by persons unique email address, if not found displays en error message, otherwise checks if friend requests exists in person friend requests list and deletes selected friend requests. Database is updated.

private HttpServletRequest challengeMonster(HttpServletRequest request, Person user):
Method sends fight request to users selected friend and updated the database.

private HttpServletRequest breedMonster(HttpServletRequest request, Person user):
Method gets users and friends selected monsters to generate new monster and updates sellers and users money, the database is updated. If not enough money to buy breed the appropriate message is displayed.

private HttpServletRequest buyMonster(HttpServletRequest request, Person user):
Method checks if user have enough money to buy a monster, if not the error message is displayed, otherwise sellers and users money is updated and sellers monster changes its owner ID to current user ID,
Monster, seller and user information's in database are updated.

public String getServletInfo()
It has no parameters, and simply returns a String "Short description". Has not been fully implemented.

4.2.7 MonsterFightServlet

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
Does nothing. Not fully implemented.

public HttpServletRequest fight(Fight fight, HttpServletRequest request)
personDAO is re-initialized and used to create two new Person objects, challenger and user. Then, the monster's are retrieved from the Fight object in the parameters. Please refer to the "Algorithms" section below for information on the fight algorithm that is in this method. After the fight has taken place, the monster information is updated and a check is made to see which monster died. If the opponent's monster died, it is removed from the database, the request attribute "fight result" is set to "You won" and user money is increased by 100. Otherwise, the user monster is removed, the attribute is set to "You lost" and user money is decreased by 100. Finally, the user and challenger have the fight offer removed and their info is updated; the request is returned.

protected void doGet(HttpServletRequest request, HttpServletResponse response)
Calls the process request method.

protected void doPost(HttpServletRequest request, HttpServletResponse response)
Used to post-process requests. Allows user to set session attributes such as current person, current monster, set up fight requests and delete said requiests.

public String getServletInfo()
It has no parameters, and simply returns a String "Short description". Has not been fully implemented.

4.3 Algorithms

4.3.1 Fight algorithm:

The algorithm for fighting between monsters is loosely based on the method for conducting fights in the game *Dungeons & Dragons*. Each round of the fight involves a dice roll by each monster – in our program it is the generation of a random double float between 0.0 and 1.0. If the dice roll is equal to 1, or if it were a 20-sided die, a "natural 20", a critical hit is scored and half of the attacker's strength is removed from the defender's health. Otherwise, the attacker's strength is added to the randomly generated float and the sum is compared to the defending monster's defence. If the sum is higher, a hit is scored and the defender's health is reduced by a quarter of the attacker's strength. Otherwise, it is a miss.
This is conducted more or less simultaneously for both monsters. At the end of the loop, a check is made to see if one of the monsters' health is below zero, if so it is removed from the database and an appropriate message is broadcast.

4.3.2 Breeding algorithm:

Firstly, an 'empty' Monster object is created, this is the resulting 'baby' Monster. Its stats are based on the stats from its parents. A dice roll (results being 0, 1 or 2) determines which stat is inherited from which parent (strength, defence or health) and which one is averaged from the two parents. For example, if the result of the dice roll is 0, then the baby inherits its strength from the first parent, its defence from the second parent and the

health is an average of the two parent's health. Finally, there's a 5% chance of random mutation. So, a random number between 1 and 20 is generated, and if the result is 1, then another number is generated that decides which stat is augmented by 25%.
Lastly, the baby monster is given a set lifespan, its name is created and birthdate set.

4.3.3 New monster generation:

Monster generation is fairly easy, it revolves around number generation using the Random class. Each stat is generated in turn. If it is below 0.5, 0.5 is added to it (giving an effective range of 0.51-1.0). A name is randomly chosen from a set list. Finally, it is given an ownerID, or the user's email address.

4.3.4 Lifespan calculation:

This is one of the simplest algorithms. It simply subtracts the difference of the current time and the monster's birthdate from the monster's lifespan. If this results in the lifespan being less than 0, the monster is removed from the database.

4.4 Main data areas

There are only two objects used in this program, those are Monster and Person objects. They contain all the information needed for the functioning of the program (e.g., monster stats, monster ownership, etc). Objects of type Monster are stored in a database file, monster.db, whereas Person class objects are stored in person.db. The Person class also contains four fields that are ArrayLists – three of these are String type, and the fourth is a custom type, 'Fight'. These four lists contain the friends, friend requests (sent and received), fights (challenges sent and received) as well as a list of all the most recent activity of that user (fights fought, monsters bought, etc) (not implemented).

4.5 Files

The only files used by our program are two objectdb files. The first one is monster.db, used for storing monster objects and their attributes – owner, various stats such as strength and their name. The second file is person.db, used for storing the person objects and any other info pertaining to people – list of friends, list of monsters, friend and fight requests, etc.
If these files are not found, new ones are created.

4.6 Interfaces

All of the servlets handle various session variables as well as POST and GET requests. However, none of the servlets implement GET, it's all done POST.

4.6.1 MyMonsterServlet

The `doPost()` method in the MyMonsterServlet handles various actions, such as setting the breed or sale offer for your monsters, selecting the current monster, as well as choosing which monster you want to use when responding to a fight request or breeding.

4.6.2 FriendsServlet

The `doPost()` method in the FriendsServlet handles various actions, such as sending friend requests, accepting or declining received requests, and challenging/buying/breeding monsters.

4.6.3 Register servlet

The `doPost()` method in the Register servlet checks that a person's email is not already in use, then adds them to the database.

4.6.4 LoginServlet

The `doPost()` method in the LoginServlet processes the user logins. It checks the password and email are correct before forwarding them to the MyMonsterServlet.

4.6.5 MonsterFightServlet

The `doPost()` method in the MonsterFightServlet processes various request actions, such as accepting a fight, refusing one, displaying challenger stats, etc.

4.7 Suggestions for improvements

Even though a complete working version of the project was created, there are always room for improvements. If there was extra time to work on it more features would be added, suggestions are below:

Deleting Friends - This is a feature a lot of "games" or projects similar to this (anything that uses "friends" tends to have. Sometimes the user does not want to keep in touch with other users anymore or there are falling outs. Due to time constraints we couldn't include this. Adding this feature would just take more time, no change to the structure needed.

Un-Registering - This was planned to be included, in case a user just wanted to remove themselves from the game completely. No need to change structure either.

More detailed combat - The fight system as it is very boring, it only shows who won the fight. Ideally we would have liked to include a step by step rundown of the fight as the algorithm was run. This would require some extra communication between the servlets and the code, this would probably not require that much change to the system.

Pictures for monsters - An aesthetic change to make the users more engaged with the game. Suggestions for this would be an algorithm that would set a random picture of a monster when it was born/created. We already had the pictures but not the time to implement it. This might take a bit more to code because it would mean adding more fields to the servlets to interact with. And we didn't attempt to use images anywhere else, so it could potentially need some more libraries to work.

4.8 Things to watch for when making changes

Generally speaking, there is not much to look out for when making changes except for the obvious. For example, files are called upon, such as `forwardURL="friends.jsp"` so renaming files will require updating their various references. Also, certain objects and/or variables are passed down between different methods or even whole classes, so care should be taken when renaming them and/or changing their type, so as to avoid an incompatible type being passed down (e.g., an `int` is changed to a `double` in one of the first methods, when it is passed down to the next method that still requires an `int` errors will occur).

4.9 Physical limitations of the program

To the best of our knowledge, the program is not very resource intensive, and so may run on a variety of systems. It has successfully run on machines in various computer labs around the Aberystwyth University campus, as well as personal machines of several group members.

4.10 Rebuilding and Testing

All of the project files for the code and configuration is automatically controlled by the IDE Netbeans, and is located in the 'Coding' or in the 'trunk' branch of the github repository called 'team-awesomer-monster-mash'.

The entire project can be loaded from the project folder and Netbeans will automatically recognise the project setup. The tests that can be run for the project can be found in the produced test specification, and the tests on the specification are passed if the actual result is equal to the expected result.

REFERENCES

[1] Software Engineering Group Projects: Producing a final report. C. J. Price, N. W. Hardy and B.P. Tiddeman .SE.QA.11. 1.6 Release

DOCUMENT HISTORY

Version	*CCF No.*	*Date*	*Changes made to document*	*Changed by*
1.0	N/A	15/02/2013	N/A - original version	PW
1.1	N/A	15/02/2013	Updated formatting and fixed few errors	PW