Fairfield University

BS Software Engineering Program

Implementation Report For SW 304

CodeFighter Application

Group 5

Samuel Nguyen, Carlos Urena, Jack Foley, Marydjina Barionette, Joey Graham

Software Engineering Methods SW401

Amalia Rusu

November 21st, 2016

***Table of Contents - CodeFighter Implementation Report***

Implementation Language…………………………… ………………………………..2

Purpose…………………………………………………………………………………..3

File Names…………………………………………………………………………….....3

File Header……………………………………………………………………………….3

Comments………………………………………………………………………………..4

Variable Names………………………………………………………………………….5

Method Arguments……………………………………………………………………...7

Parentheses, Braces, and Indentation………………………………………………..7

Control Structures……………………………………………………………………….9

**Implementation Language**

The CODEFIGHTER team used Unity to implement the application. The team decided that using Unity would be the most practical way of implementing an application that was truly cross-platform. Unity offers a wide-range of supported devices, including iOS, Android, Windows, Mac, and much more through the same source code. Cross-platform compatibility allows CODEFIGHTER to reach a larger audience, and be much more flexible for end users.

Another reason Unity was used is because it offers integrated multiplayer and online database support. Unity also has much more resources and tutorials than other engines like Unreal and Cryengine. It primarily uses C# as its programming language, but it offers the option of using other scripting languages like JavaScript.

Unity, along with Microsof Visual Studio, offers a blend of robust coding environments and powerfult 3D tools, that make the game making process streamlined and smooth.

***Coding Conventions for CodeFighter Application***

**Purpose:**

To give a clear and concise description of the coding conventions that are used in the application.

**File Names:**

Unity uses several different types of files, and they all have different suffixes to differentiate them.

Files using the suffix *.cs* are C sharp files.

Files using the suffix *.csproj* are C Sharp projects in Unity.

Files using the suffix *.ini* are INI configuration files.

Files using the suffix *.txt* are text files.

Files using the suffix *.meta* are graphical metafiles.

Files using the suffix *.anim* are Unity animation files.

Files using the suffix *.prefab* are Unity prefab files that store a GameObject object complete with components and properties.

Files using the suffix .sln contain text-based information the environment uses to find and load the name-value parameters for the persisted data and the project VSPackages it references.

**File Header:**

Each unity script file will begin with a header as shown below which will state the filename, purpose, author(s), revision version, and date of the last revision.

In order to make the header into a comment, two slashes are placed in front of each line.

For example:

// File name: Login.cs

// Purpose: To create a login page for the user

//

// author: Name1, Name2

// version: Revision: 1.1

// Revision Date: 11/10/16 15:22:43

**Comments:**

In the source code there are C Sharp comments. A C Sharp comment consists of two slashes before the line of text that is intended to be a comment. These comments serve the purpose of improving readability in the code and making the code easier for others to edit and modify in the event that the original programmers are unavailable to do so. All of are scripts are programmed in C Sharp so all comments are structured as follows:

***Example of a comment is a Class File:***

***// Destroy(this.gameObject) Added to end game***

**Variable Names:**

The public float **startSpeed** decides the starting speed of the enemies. It was named in this way to make its meaning very obvious.

The public float **speed** decides the current speed of the enemies. It was named in this way to make its meaning very obvious.

The public float **health** decides the health of the enemies. It was named in this way to make its meaning very obvious.

The public integer **worth** decides the worth of the enemies. It was named in this way to make its meaning very obvious.

The float **amount** decides the amount of damage that the enemies take. It was named in this way to make its meaning very obvious.

The float **pct** decides the percentage of the startSpeed the enemies are moving at. It was named in this way to make its meaning very obvious.

The static integer WaveMax indicates the maximum number of waves for each level. It was named this way to make it very obvious and easy to understand.

The boolean variable **gameEnded** indicates whether the game has been ended, and it was named this way to make it easy to understand.

The public string **u** is the string representing the correct username, and it was named that way because it is the first letter of the word “username”.

The public string **p** is the string representing the correct password, and it was named that way because it is the first letter of the word “password”.

The string **usernameString** represents the username that the user submitted, and it was named that way because it is obvious and easy to understand.

The string **passwordString** represents the password that the user submitted, and it was named that way because it is obvious and easy to understand.

The public input field UserField represents the input field where the user inputs the username, and it was named that way to make it easy to identify.

The public input field **PassField** represents the input field where the user inputs the password, and it was named that way to make it easy to identify.

The public static integer **Money** indicates how much in-game currency the player has, and it was named that way so that it would be easy to understand.

The public static integer **startMoney** indicates how much in-game currency the player starts with, and it was named that way so that it would be easy to understand.

The public static integer **Lives** indicates how many lives the player has, and it was named that way so that it would be easy to understand.

The public static integer **startLives** indicates how many lives the player starts with, and it was named that way so that it would be easy to understand.

The public boolean variable **panelOpen** indicates whether the question panel is open and displaying a question. It was named this way because it is easy to understand.

The string **sceneName** tells us which scene goes after the previous one. It was named this way because it would be easy to understand.

**Method Arguments:**

In order to call a method, we must also include arguments that give the method the information that it needs. If the method being called requires a string input, then we will use a string argument, if it requires an integer input, we will use an integer argument, and so on

.

We have data type and object arguments. We must fulfill these parameters for consistency and for proper functionality. These also serve to allow proper interaction between the states of the game.

For example:

**public void TakeDamage(float amount)**

This method requires a float input, so we satisfy it by passing the float “amount.”

**SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);**

This method requires the result of another method as its argument, so we pass it that method.

**public void BuildTurretOn (Node node)**

This method requires a Node argument, so we pass it the Node variable node.

**Parentheses, Braces, and Indentation:**

**Parentheses** are placed after the method name or a loop name.

void Awake ()

{

if (instance != null)

{

Debug.LogError("More than one BuildManager in scene!");

return;

}

instance = this;

}

Opening **braces** are placed one line below the method name or loop statement. The closing brace is placed after the content, on an even orientation.

if (instance != null)

{

Debug.LogError("More than one BuildManager in scene!");

return;

}

**Indentations** are used to nest elements of the code. Every element that is a part of another element is indented to display the relationship between the two. This allows programmers to see which elements are a part of other elements.

**Control Structures:**

**For loops**

Multiple & Single statements

for(int i = 0; i < tabs.size(); i++)

{  
 statement1;  
 statement2;  
}

**If Statements**

if (condition)

{  
 statements;  
}

Else

{  
 statements;  
}

**While Loops**

No While Statements Were Used

**Switch Statements**

No Switch Statements were used