SOFTWARE DEVELOPMENT PROJECT: Beyond Hangman

Jordan Gorrell

February 1st, 2019

Contents

1 | Revision History

Date	Version	Description	Author
8/2/19	1.0	Project Plan and Skeleton Code of classes	Jordan Gorrell
22/2/19	2.0	- UML Documents Created + Class Diagram + Use Case Model + State Machine Diagram - Round Class and User Class implemented	Jordan Gorrell
8/3/19	3.0	- Unit Testing of existing code - Test Report on the Unit Testing - No new code added, only the testing	Jordan Gorrell
21/3/19	4.0	- Implemented the classes: + Game + LivesGame + NormalLivesGame + HardLivesGame + MultiplayerGame + UsernameDataHandler - Updated UML documents to reflect the way the applicated has been implemented. + Use Case Model + State Machine Diagram + Class Diagram - Unit Test and Report done on newly implemented code	Jordan Gorrell

2 | General Information

Project Summary	
Project Name	Project ID
Beyond Hangman	jg222zr_1DV600
Project Manager	Main Client
Jordan Gorrell	Book-lovers, English Majors/Graduates

Key Stakeholders

- Designer/Programmer
- End-Customer/Main Client

Executive Summary

Beyond Hangman is a computer version of the classic pen and paper game of Hangman, where a player tries to guess a hidden word by guessing individual letters. This project is being done to create a high-difficulty yet enjoyable word-game experience for those interested in both tough challenges and the English language.

3 | Vision

The user will start the program and be asked to enter their name if it is their first time using the program. If they are a returning user, they will choose from a list of previously entered names. The names hold statistics associated with the user. As of now there is not a plan for a password.

After entering a name or choosing an existing one, the user can pick between single-player and multi-player. Multi-player will be essentially be no different than playing Hangman regularly with a pen and a piece of paper. Player #1 will enter a word or phrase, then the program will ask the player to confirm that they have entered their word/phrase correctly. Afterwards, Player #2 will guess letters. There will be an image of the hangman showing how close they are to losing, a row of letters that have been guessed, and the word/phrase. The word/phrase's hidden characters will be represented by underscores. The revealed letters replace the underscores when discovered. These letters will be capitalized.

After the round is over, regardless of the outcome, the roles of the players switch, another round is played, and repeat. Each player has five turns guessing. The one who guesses more words correctly than their opponent wins.

Single-player will use the same gameplay style as multi-player. Here, however, the computer selects words from a specific word bank based on the difficulty selected by the user. Normal and Hard are currently the only two difficulties planned. The user has three lives. If they get a word/phrase wrong, they lose a life. The goal is to complete as many words as possible before losing the three lives. This is the mode that saves user statistics. No statistics are saved from multi-player. Statistics saved are unique for each difficulty: number of games played, most words ever solved in one game, number of words solved all time, number of words failed all time. There will also be leaderboards.

REFLECTION | Writing this vision document has been more satisfying than I thought it would be. Having the full, basic vision written down somewhere instead of having just a few scribbled notes and thoughts in my mind already makes it feel more concrete, even though nothing has been coded yet. Having this to refer to takes some stress away from trying to remember everything that was planned. It is also easy to take these as specifications (what will be done) and directly move on to design (how to do it).

4 | Project Plan

4.1 Introduction

A one or two player game based off the simple paper and pen game of Hangman. Single-player wise the game focuses on difficult words chosen to give a challenge to players that are late teens or adults. This Project Plan section covers what decisions are being made, why they are being made, and what is being used to bring this project to life.

4.2 Justification

Beyond Hangman is being created to cater to people who enjoy other difficult word-related games or activities (such as crossword puzzles, for example). The target players of Beyond Hangman are generally 16 or older and enjoy reading or studying English.

4.3 Stakeholders

- Designer/Programmer
    - Same person. Makes all design related decisions and implements those decisions. The designer is putting an emphasis on having a user friendly interface.
- End Users
    - These are book-lovers, English Majors, others who enjoy difficult word-based games.
        - These stakeholders affect how the game is designed and implemented. Since they are generally knowledgeable of complex English words, they affect the difficulty level that the game will be set at.

4.4 Resources

- Computer: Acer Aspire E 15
- IDE: Eclipse using Java8 and JUnit
- Software Engineering - Ian Sommerville, 10th Edition
- Lecture materials on Software Development
- Access to Online Resources
- Approx. 2 months for the developer to learn the relevant aspects of the design process and implement the game.

4.5 Hard- and Software Requirements

Developer: Java 8

End-User: JRE 1.8.0_191

The application is tested on the above JRE. It is therefore recommended for users to run it on the same JRE for maximum stability. The user can choose to use an IDE, such as Eclipse, to run the program, or run it directly in the command prompt.

4.6 Overall Project Schedule

Due Dates:

1. Friday, February 8th, 11:55pm
    a. Project Plan
    b. Skeleton Code of Classes

2. Thursday, February 21st, Noon
    a. Some Gameplay Implemented
    b. Use Cases and UML Diagrams
        i. State Machine
        ii. Use Case Diagram
        iii. Class Diagram

3. Friday, March 8th, 11:55pm
    a. Test Report of Existing Code

4. Thursday, March 22nd, Midnight
    a. Finished Implementation
        i. Game-Modes
        ii. UsernameDataHandler
    b. Updated Diagrams
    c. Test Report on New Code

4.7 Scope, Constraints and Assumptions

Scope:

Beyond Hangman will include three game-modes, two of which are single-player, the other is two-player. Both of the single-player modes do essentially the same thing, but with differing levels of difficulty. The player will try to guess as many words correctly before they have been hanged three times. Both modes are difficult, it is simply that one is even more difficult than the other.

Players will have a username associated with them that they either register if it is their first time playing. Otherwise, they select their username from a list of existing usernames. The usernames have statistics associated with them that are stored/updated after they are done playing the game.

Two-player games will consist of the players taking turns picking a word for the other player to solve. If a player guesses the word that the other player has picked for them, they get a point. The player with the most points by the end of the game wins.

Out-of-scope are things like game-modes that have easier words. These are omitted because they do not cater to the target-group. Also, usernames will not have passwords associated with them. This is because the goal is for the user to be able to play the game quickly and easily. The statistics are not so valuable that passwords are necessary. Statistics are rather just a fun side thing that the user may view if they so desire.

Constraints:

The statistics for the game are stored in a .txt file. Access to a more secure means of storage is not feasible for this project. This means, however, that savvy users will be able to locate this file and adjust their stats as they desire. Seeing how stats do not affect gameplay in any way (they are nothing more than numbers for the user to look at), this is acceptable. There will also be no installer set-up for Beyond Hangman. This means that, as stated in section 4.5, the users need some version of JRE (preferably 1.8.0_191), and run the program in an IDE such as Eclipse or run it in the command prompt.

Assumptions:

It is assumed that the users are English speaking and further assumed that they have a good grasp of the language, thus the difficulty level of the game. It is also assumed that users have the ability to set-up the necessary environment to run the program.

Reflection | Writing the information for sections 4.1 - 4.7 has been tedious but worthwhile as it further solidifies the vision and plan for the application. Focusing on the target-group is greatly helpful for narrowing down exactly what it is that should be achieved. Writing the scope has been particularly helpful in that it not only helps one further understand what is happening, but also why things are going to be done the way that they are going to be done.

5 | Iterations

5.1 Iteration 1

The goal of this iteration is to get this document up-to-date and writing skeleton code. This means filling out general information in section two, writing the vision in section three, writing and answering questions in the project plan section four, doing analysis, and filling out the time log. Also necessary is implementing the skeleton code for the project. Lastly, the time-log needs to be analyzed.

The skeleton code will essentially be empty classes, and in the following iteration a class diagram will be created to document the relationships between all the classes. Each of the bullet points below represent a class:

- User
- Round
- Game
    - LivesGame
        - NormalLivesGame
        - HardLivesGame
    - MultiplayerGame

The bullet points are set-up this way to indicate where a class is a subclass of another class. Here, for example, **LivesGame**, is a subclass of **Game**, and the superclass to **NormalLivesGame**. Also, the **Game** class should be abstract.

5.2 Iteration 2

The goal of this iteration is to write Use Cases, UML diagrams, and further implement the program.

- Write Use Cases
- Create Use Case Diagram
- Create Class Diagram
- Create State Machine Diagram
- Implement Interface Functionality
- Implement functionality for a *Round* of Hangman
- Implement functionality for the *User* class

The *Round* class simulates a round of Hangman. This is where the player guesses letters, hoping to solve the word. As this is a console application, users will give input through the console via a scanner object. A round is over when the player successfully reveals the word or is ‘hanged’ (unable to figure out the word). A *Round* object accepts a word as a parameter, as well as a User object if the game

is single-player in order to know which user's statistics to update. The *Round* object takes the word, hides the letters (represented by underscores), and displays the noose and hanging individual, so the player can see how many more chances they have until they lose. Multiplayer mode does not keep track of statistics and therefore does not need the User object as parameter. This means two constructors are needed: One that takes a string (the word), and a user object, and another constructor that takes only a string.

The *User* class stores individual user statistics. For each single-player difficulty (normal or hard), certain statistics are saved:

- Number of games played in each difficulty
- Number of games played regardless of difficulty (as in, total games)
- Most words ever solved in one game (one stat for each difficulty)
- Number of words solved all time (rounds won, regardless of difficulty)
- Number of times hanged (rounds failed, regardless of difficulty)

This means there will be a total of seven statistics saved. Users and their data will be saved and repeatedly accessed and updated as needed. Given this information, the *User* class needs an integer field for every stat, as well as a string field which stores the user's username. Functions for updating the statistics fields also should be put in place.

5.3 Iteration 3

The goal of this iteration is simply to test the existing code. No new features will be added for this iteration. A test report will be produced during this iteration to document how well the testing went.

5.4 Iteration 4

The goal of this iteration is to complete the implementation for Beyond Hangman, update/adjust/create new or existing UML documents in order to make all the documentation reflect the way the application has actually been implemented. Also, tests should be run on newly implemented code and a test report should be produced.

- Implement the Game-Modes
    - Game
        - LivesGame
            - NormalLivesGame
            - HardLivesGame
        - MultiplayerGame
- Implement UsernameDataHandler class
- Update Use Cases and Use Case Diagram
- Update State Machine
- Update Class Diagram
- Test New Code, Make Test Report

The abstract class *Game* will have a single method that verifies that a word is a valid Beyond Hangman word. In Beyond Hangman, a word/phrase can only consist of letters (no more than two of the same letters consecutively) and spaces (no more than one consecutive space). It also should contain at least 5 letters and should not exceed 40 total characters.

The *LivesGame* class is a subclass of the *Game*. A *LivesGame* object takes a *User* object as a parameter, in order to save statistics. A LivesGame object has an integer field numOfLives which indicates how many times a player can lose a round before the game is over. It also has a field ArrayList<String> wordBank which is the list the that holds the words that will potentially be played. The difference between the *NormalLivesGame* and *HardLivesGame* classes is the contents of their wordBank. Both classes are subclasses of *LivesGame*. These have hardcoded wordBanks and numOfLives (set to 3). The *HardLivesGame* class will in theory have a wordBank of words/phrases that are more difficult to solve than the *NormalLivesGame* class.

When a user chooses playing either of the *LivesGame* types, they solve words until they run out of lives. The word is selected randomly from the wordBank. A variable tracks how many words have been solved this game, and the appropriate user statistics are incremented and updated at the end of every round and every game.

MultiplayerGame is a subclass of Game. The *MultiplayerGame* class consists of ten rounds, and ends either after these ten rounds are over, or until one player has a large enough lead that they cannot be tied. The game consists of two players. The program asks Player #1 to enter a word or phrase and confirm the word/phrase. The word/phrase will also be verified/validated. Once verified, this word/phrase will be entered as a parameter to a new Round object, and a round of Hangman is played by Player #2 using the word entered by Player #1. If Player #2 succeeds with the round (successfully solving the word), they get a point. Then the roles switch. The player with the most points at the end of the game wins. A method will check after each round if either player has a lead that can no longer at least be tied, and if so, the game ends early.

The *UsernameDataHandler* class handles the storage and management of statistics. It saves data to and reads data from a file called ‘data.txt’. When starting the application this class generates the list of existing users for the user to pick from by parsing through the data.txt file. Instead of a leaderboard, as initially mentioned in the vision, users will only be able to look at their individual statistics. This class handles the printing of these statistics.

In a separate package, the *PlayGame* class will be where the application is run from. This class handles the out-of-game interface and interaction with the user. When executing the program, the user should initially be able to pick their username from a list of existing usernames of be able to register a new one. After selecting a username, a menu should be displayed which allows the user to decide on an action. These actions are:

- [1] Play Normal Mode
- [2] Play Hard Mode
- [3] Play Multiplayer
- [4] Your Stats
- [5] Quit

The user entering ‘1’ should start a game from the *NormalLivesGame* class, ‘2’ a game from the *HardLivesGame* class, ‘3’ a game from the *MultiplayerGame* class, ‘4’ should prompt the *UsernameDataHandler* class to display the user’s statistics, ‘5’ should allow the user to quit the application.

6 | Risk Analysis

6.1 List of risks

Risk	Probability	Impact
Underestimating Time Required	Moderate	Moderate
Focus Placed on Wrong Things	Moderate	Moderate
Computer Dies	Low	“Catastrophic”
Family Emergency	Low	High

6.2 Strategies

Underestimating the time required for the project has a reasonable likelihood of happening, and its impact could be anywhere from little to a lot. This will be dealt with by instead allotting more time to tasks than they look like they will take at first glance. For this project, overestimating time required will not cause any problems, and will protect the project from unnecessary issues.

Having the **focus placed on wrong things** is a problem that happens with projects like these, where endlessly implementing more features sounds better than doing documentation and testing. Having this project plan should help combat that.

The **main computer dying** is another risk, but can be dealt with by backing up the computer online.

A **family emergency** has no strategy.

Reflection | With other projects, I am sure that the risks will be more concrete and clearer. Here I had to reach a bit to just make something. Nonetheless, they are legitimate risks to my Hangman project. When it comes to misunderstanding instructions, I may have already done so. Am I supposed to write stuff in the Iteration section? Was I supposed to write a huge project plan in the section that I wrote it? Or was it supposed to be spread out between the questions 4.1-4.7 somehow? I can not really see how that would work, so I think I did it properly. **Update: Wrong!**

7 | Time logs

Iteration 1:

Objective	Estimated Time	Actual Time
Vision	1:00h	1:30h
Project Plan	1:00h	1:30h
Risk Analysis	15m	30m
Skeleton Code	30m	30m

Time-Log Analysis: Both the vision and the project plan took slightly longer than expected. In thought it sounds like it shouldn't take much time to write a page (Vision) or answer some questions (Project Plan), but in reality there is a lot of time spent just thinking and making sure that what is written actually makes sense. The reasoning is the same for the risk analysis.

Iteration 2:

Objective	Estimated Time	Actual Time
Use Case	2:00h	4:00h
State Chart	30 minutes	5:00h
Class Diagram	1:00h	30 minutes
Implementation	4:00h	3:30h

Time-Log Analysis: This is the first time I have worked with UML and really making diagrams like this, so I severely underestimated things. I made my state chart cover more things than I expected when I wrote the estimated time, and it also took a very long time working my way through the logic of the diagram to make sure I was doing it well enough. Those two things combined made for a very large difference between the estimated time for a state chart and the actual time.

Iteration 3:

Task	Estimated Time	Actual Time
Write Manual TC	2h	2.5h
Write/Run Unit Tests	3h	5h
Running Manual Tests	30m	15m
Test Report	1h	3h

Time-Log Analysis: Writing manual test cases was only off by thirty minutes, and considering it was my first time writing them, that's not bad. The Unit tests took quite a while and longer than expected. Finding methods suitable for testing was the first challenge, and then coming up with the appropriate tests. It simply took longer than expected to come up with acceptable tests. The test report also took time as there were more aspects to add to it than I was originally aware of.

Iteration 4:

Objective	Estimated Time	Actual Time
Update Use Cases/Diagram	45m	30m
Update State Machine	30m	15m
Update Class Diagram	45m	1:15h
Implement Remaining Features	6:00h	9:00h
Manual TC for New Code	2:00h	3:00h
Unit Tests for New Code	2:00h	1:00h
Test Report for New Code	2:00h	1:00h

Time-Log Analysis: All in all, the estimated time totals up to 14:00h, and the actual time total was 16:00h. Considering the amount of hours that is, being two hours off is not too bad at all. When it comes to implementing the features, little things continued to pop up that I had not previously realized that I would have to address, resulting in a lot more time spent there than expected.