SOFTWARE DEVELOPMENT PROJECT TEMPLATE

Jordan Gorrell

February 1st, 2019

Contents

1 | Revision History

Date	Version	Description	Author
2/22/19	2.0	Updated Code and Timelog	

2 | General Information

Project Summary	
Project Name	Project ID
Project Manager	Main Client
Key Stakeholders	
Executive Summary	

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 easier to implement than single-player and 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

Project Structure

This Hangman project will consist of eight classes, including one Main class that contains the main method which will execute the program. The other classes are:

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

The **User** class stores individual user statistics. These statistics are as identified in the vision document. For each single-player difficulty (normal or hard), certain statistics are saved:

- Number of games played in a difficulty(this is a whole game, not just a round)
- Number of games played regardless of difficulty
- Most words ever solved in one game
- Number of words solved all time (rounds won)
- Number of times hanged (rounds failed)

This means there will be a total of nine statistics saved, as every stat is difficulty dependent aside from the total number of games played. Users and their data will be saved and repeatedly accessed and updated as needed.

The **Round** class simulates a round of Hangman. This is where the player guesses letters, hoping to solve the word. 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.

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. A *MultiplayerGame* object takes a Scanner as a parameter. The game consists of two players. Using the Scanner, the program asks Player #1 to enter a word or phrase and confirm the word/phrase. The word/phrase will also be verified/validated. 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). A method will be made for this in the abstract class **Game** (this will be the only thing in this class). MultiplayerGame is a subclass of Game. 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, they get a point, no point if they lose. 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 be tied, if so, the game ends early.

The **LivesGame** class, a subclass of the *Game* class, was originally named SinglePlayerGame, but in case other single-player game modes are thought of, the name was changed. A *LivesGame* object takes a *User* object as a parameter, in order to save statistics. A *LivesGame* object has a field int 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.

If everything is coded properly and in time, a leaderboard feature could also be implemented.

Running the Program (User Case)

1. Player chooses existing profile or creates new one
2. Player selects game mode;
    a. NormalLivesGame
    b. HardLivesGame
    c. MultiplayerGame
3. Plays until game ends. Runs out of lives for LivesGame or until a player wins (or there's a tie) in multiplayer
4. Player is returned step 2 or quits the program or looks at their statistics or the leaderboards.

Reflection | Similar to the vision document reflection. Having this written down now just further solidifies the project in my mind. From this it would be a breeze to write the skeleton code for the entire project. It is also open-ended enough to adjust to changes that may be desired, as well as adding

additional game features, such as additional game modes. It is a secure feeling to have a plan.

4.1 Introduction

A one or two player game based off the simple paper and pen game of Hangman.

4.2 Justification

To learn about the process of creating Software, especially the parts that are not simply coding. Also, achieve a slightly decent final grade.

4.3 Stakeholders

Myself and perhaps the instructors.

4.4 Resources

Computer, Eclipse, Java, JUnit, Textbooks

4.5 Hard- and Software Requirements

Eclipse, Java/Java Compiler/JVM/JDK

4.6 Overall Project Schedule

Iteration Due Dates:

1. Friday, February 8th, 11:55pm
2. Thursday, February 21st, Noon
3. Friday, March 8th, 11:55pm
4. Week 12

4.7 Scope, Constraints and Assumptions

Detail what is part of the project and what is outside – specify the scope of the project.

5 | Iterations

Plan for four iterations, including this. This is a fine-grained plan on what is to be done in each iteration and with what resources. To begin with, this is a plan of what we *expect* to do, update this part with *additions* (never remove anything) when plans do not match up with reality. Also make time estimates for the different parts.

In this course the overall planning has in some ways already been decided, so use the template to provide more details on specific tasks that define *your* project. Remember that you can plan to add features to any of the phases as long as the main focus is also met.

The first assignment is to complete iteration one.

5.1 Iteration 1

Fill out the majority of this document. Write the skeleton code based on the classes proposed in the project plan.

5.2 Iteration 2

In this iteration you need to add some features to the game *but* after you have first modelled them using UML. All diagrams need to be included in the project documentation and should be implemented in the way modelled.

5.3 Iteration 3

You may include additional features to the game in this iteration, but the main focus is on *testing*. Plan, perform and document your tests in this iteration.

5.4 Iteration 4

The outcome of this iteration is *the complete* game. Reiterate the steps in iteration 1 – 3 for a set of new features but also remember to see the project as a whole, not only its parts.

6 | Risk Analysis

All projects face risks that make it important to prepare for what might happen. Use the chapters in the book as well as the content of the lectures to identify the risks within this project. As always, write down your reflections on creating a risk analysis. This reflection should be about 100 words.

6.1 List of risks

Risk	Probability	Impact
Procrastination	High	Moderate
Misunderstanding Instructions	High	Moderate
Computer Dies	Low	“Catastrophic”

6.2 Strategies

When it comes to **procrastination** for me, it is often a matter of not having a plan. Planning specifically what to accomplish on a given day, rather than just “working on the project,” will give me more direction and motivation. Solving the of **misunderstanding instructions** should be as simple as consulting the teachers. In the case of my **computer dying**, I am probably asking for it. Nothing is backed up. Maybe I should do that.

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 little 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.

7 | Time log

Assignment 1:

Objective	Date	Estimated Time	Actual Time
Project Planning	February 4th, 2019	3:00h	cont
Project Planning Continued	February 5th, 2019	“ “	3:30h
Skeleton Code	February 6th, 2019	30 minutes	30 minutes

Assignment 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

8 | Handing in

All assignments have a number of files to hand in. The overall advice is to *keep it simple*. Make it easy for the receiver to understand what the files are by using *descriptive* file names. Use as *few* separate documents as possible. Always provide a *context*, that is *do not* send a number of diagrams in “graphics format”, but always in a document where you provide the purpose and meaning of the diagrams. Remember that the “receiver” is in reality a customer and as such has very little knowledge of the diagrams and documents – always provide context that make anything you hand in understandable to a non-technical person.

To hand in an assignment, make a git release and hand in the link via Moodle to that release.