# *Programming III (420-G30-HR)*

# *Assignment 2 – Hangman Using Linked Lists*

Date assigned: Wednesday, Sept 27

Milestone 1: Test Cases and Class Design: Tuesday, Oct 10, 8am

Milestone 2: Complete Functionality Monday, Oct 30, 9am

***Objectives:***

After completing this assignment, the student should be able to:

1. Draw a class diagram.
2. Design JUnit test cases for a linked list.
3. Use a singly linked list.
4. Use a doubly linked list.
5. Write a serialized object.
6. Read a serialized object.
7. Override the **toString()** and **equals()** method of the **Object** class.

***Problem Specifications:***

Develop a program to play the game Hangman. You must read a list of words (or word phrases) from the file **word\_db.txt**. A sample of this file is provided on Moodle; its format is a single word sequence per line. The Hangman program should **randomly** select a word to use; this is the word that the player is trying to guess. The game displays placeholders (i.e. ‘\_’) for each guessable letter in the word. The player then guesses one letter at a time. If the letter is in the word, then **all** occurrences of that letter should be displayed to the player. If the letter is not in the word, then the letter should be added to the list of guessed letters for the user to see, and a mistake is recorded. If the player guesses the word before six mistakes are made, they win. If they don’t then they lose, and the answer should be displayed to the player.

The program should ask for the player’s name at the start of a game. Past player names should be available in a **drop-down list** to select from. The program keeps track of the number of games each player has played. At any time, the player can view the scoreboard, which shows the name of each player, in alphabetical order, and the number of times each player has played the game and the number of wins each player has had.

The player has the option of asking for a hint. This should show a random letter in the word to the player. (Note – if the letter ‘a’ is given as the hint, and there are three letter ‘a’’s in the word, then all 3 occurrences of the letter ‘a’ or ‘A’ should be shown.

Once a word has been used in a game, it must not be used again. This does not mean that the word should be deleted from the **word\_db.txt** file; this file must never be modified. The word should be deleted from the internally stored list.

A game can be exited at any time, and when the program is restarted, you must continue from where the game left off. (This means that you only read the **word\_db.txt** file the very first time you start a game; all subsequent times the state will be saved). You should preserve the player scoreboard and the current game across a program exit. **Serialization** must be used to accomplish this save of state.

In addition to the notes provided, here are two good references for learning about serialization:

<http://www.tutorialspoint.com/java/java_serialization.htm>

<https://www.geeksforgeeks.org/serialization-in-java/>

The user should be able to perform the following functionality at any time, from a **drop down menu**:

1. Start a new game, even if a game is in progress
2. View the scoreboard, without affecting the current game in progress
3. Get a hint. This should show a random letter in the word.
4. Quit the program, even if a game is in progress
5. Learn about the rules of the game

Note: You will be using your PEA GitHub account and invite me (and only me) to your project. You will need to demonstrate your GitHub project is setup and populated with your code. You must have at least two code updates to your project.

Your design must use the following:

* 1. Your frame design is up to you. There are 10 marks allocated for the frame design. A basic frame that does not have any colour or creativity will receive a mark of 5/10.
  2. For singly linked lists, use the provided **SLNode** and **SinglyLinkedList** code. For doubly linked list, use the provided **DLNode** and **DoublyLinkedList** code.
  3. A singly linked list must be used to store the words read in from the dictionary.txt file. When a word has been used, it must be deleted from this linked list.
  4. A singly linked list must be used to store the letters of the word to be guessed. You cannot use string processing – you must use a singly linked list of characters.
  5. A singly linked list must be used to store the guessed letters.
  6. A doubly linked list must be used to store the scoreboard. The scoreboard must be kept in alphabetical order.
  7. Serialization must be used for the file reading and writing of the program state.
  8. Make sure that you keep your frame logic separate from the program logic.
  9. Junit test cases must be written for all player and scoreboard functionality.

***To Do:***

***Milestone 1:*** *Initial Design, Test Cases and Class Design*

1. Sketch a frame design for the program. This can be hand drawn, and either scanned or a legible photo of it can be submitted.
2. Draw a class diagram for the program. This is your initial draft of the class diagram showing the basic classes, containment, associations, and hierarchy. I just need to see the basic public methods.
3. Design your test cases. **Provide the rules and the equivalency partitions**. I do not need to see the sample data mapping or the test scenarios tables. This should give us an idea of the areas you are covering with your testing. (i.e. just do the first two steps of the test design process).

***Marking Scheme:***

|  |  |  |
| --- | --- | --- |
|  | **Mark** | **Out of** |
| Milestone 1: initial design, test cases and class diagram |  |  |
| Layout sketch |  | 8 |
| Class Diagram & Class design |  | 10 |
| Black box test case design (rules, equivalency partitions) |  | 15 |
| Milestone 2: Complete Functionality |  |  |
| Basic Frame code – functionality matches requirements; |  | 15 |
| Frame code innovation – use of colour, creativity |  | 10 |
| Basic Game logic code – dictionary file processing, correct list types used for word and guessed letters;  Efficiency and robustness |  | 35 |
| Hint functionality – frame and logic |  | 10 |
| Scoreboard functionality – frame and logic – doubly linked list used; kept in alphabetical order |  | 15 |
| Game state saved using serialization |  | 15 |
| Junit test case code – Game Logic |  | 20 |
| Correct Program Execution |  | 10 |
| GitHub setup and shared correctly (milestones included) |  | 8 |
| Organization |  | 5 |
| **Total** |  | **176** |

***Organization Marks:***

Marks will be given for organization. This includes:

* naming files and folders according to the department standards
* giving meaningful names to variables, classes, objects, and methods
* formatting and indenting Java classes using the Eclipse format tool
* submitting the assignment in correctly on **Moodle**
* including all required files in the submitted assignment folder

***To be handed in:***

The following files should be included in a ***username*\_G30\_A02\_Linked\_Lists** folder and uploaded to **Moodle**:

1. The test cases.
2. The class diagram for the assignment.
3. The frame design.
4. The java project for the assignment. The project name should be ***username*\_G30\_A02\_Linked\_Lists**.
5. This assignment must also be stored in your PEA GitHub. The only reason I want both GitHub and Moodle submission is to officially track the date of submission.

# *Appendix A - Hints – Hangman Using Linked Lists JUnit Tests*

\*Note that these are an example of the things that you should test for your Junit tests. You may have done your Scoreboard class differently, which is completely fine, but you need to have Junit tests to thoroughly test that class. It’s hard to imagine your Player class being much different from this example though.

Player Class

* name: String
* numberGamesPlayed: Integer
* numberGamesWon: Integer
* default constructor
* constructor with Name
* accessors and mutators for each of the 3 instance vars

Scoreboard Class

* doublyLinkedList of Players (kept sorted by name)
* numPlayers: Integer
* default constructor
* addPlayer(name)
* gamePlayed(name, winOrLose:Boolean)
* Player getNextPlayer(index) (for displaying in Frame)

Dictionary (word list) Class

* Determine your rules for the min/max length of input word
* Min number of guessable characters
* Be able to filter out non-guessable characters for the initial clue display
* Invalid words/phrase lines in the input file are skipped and processing the rest of the file resumes.

Junit Player tests

1. Test constructor; create Player and make sure each instance variable is set to defaults for numGamesPlayed and numGamesWon, and that name is set.
2. Test sets – increment gamesPlayed and gamesWon and make sure they return the right value

Junit Scoreboard Class

1. Test default constructor – no players added, numPlayers = 0
2. Test addPlayer – create 1 and make sure you get it back. (boundary test)
3. Test addPlayer, more than 1 player in list and make sure you get it back (boundary test)
4. Test gamePlayed with a win, and with a loss
5. Test gamePlayed with a player that is not found
6. Test getNextPlayer with an empty list
7. Test getNextPlayer with a list of 1
8. Test getNextPlayer with a list of 3

# *Appendix B - Hints – Project Management*

Consider setting your own intermediate milestones.

Identify all your goals, and sequence them in an order that makes sense to you.

i.e. Get it working with just a hard coded phrase,

then figure out how to get the phrases from a file,

then figure out how to make it serialized/stored,

then figure out how to do the scoring