Aaron Ennis

Oregon State University

CS 162 – Introduction to Computer Science II

Final Project Design Document – Milligan’s Island

# Overview

Milligan’s Island is a text-based adventure game set on a deserted island. The shipwrecked player must investigate different areas of the island and collect items to repair the ship’s hull before nightfall and high tide when the hull will fill with water and sink the boat for good. The areas of the island are represented as derived class objects of an abstract base class (Space). Each area of the island is linked to others around it by way of pointers to other Spaces. Each Space also contains unique features that the player can interact with in order to locate and collect items that will help achieve the goal of safely leaving the island, however, only two items can be carried at once. There are only 15 hours of usable daylight and low tide conditions (represented by 15 turns) for the player to leave the island alive.

# Narrative

The player is awakened by the sunrise on a beach that stretches for miles in each direction. The player’s wrecked ship is accessible from the beach, but there is a gaping hole in the ship’s hull. It is approaching low tide, but it is also evident that the next high tide will cause water to enter the breach in the hull, filling the ship and stranding the player for good.

# Program Flow

A detailed flow of the program can be found in [Appendix A](#_Appendix_A:_Program). Upon launching the program, the user will be asked to enter the number of fighters. This number will be used for both teams to ensure an equal number of starting fighters. For each fighter slot, the user will be asked to choose the fighter type and provide a name for the fighter. This will be repeated for both teams. Once all fighters have been entered, the tournament will begin. The first fighter from each team will be pitted against one another and fight per the detailed flow show in [Appendix B](#_Appendix_B:_Fight). Combat is over when a character’s strength is depleted. A summary of the combat will be displayed. The winning team will be awarded points, and the losing team will lose a certain number of points. The losing fighter will be placed in a loser list. The winning fighter will be returned to the back of the team queue, and will regain a certain percentage of strength. The next fighter from each team queue will then fight in the same manner. The process will repeat until one team has no fighters left in queue, signaling the end of the tournament. The tournament results will be displayed, and the user will be given the opportunity to display the loser list. The user will then be given the opportunity to play the tournament again.

# General Implementation Approach

This program will be implemented using several classes: an abstract Character class that is used to derive inherited classes representing the several character types, a Queue class to hold each team, a DubList (doubly-linked list) to hold the losers, a pre-existing Die class that creates dice objects to simulate the attack and defense attributes of the Character objects, and a Tournament class that is used to initialize teams and control the program flow.

# Character Class

The Character class is an abstract class that encapsulates most of the data members and member functions needed for the derived character classes. A UML diagram of the Character class, and its derived classes is shown below. The items in red have been added to this class to accommodate the additional requirements for the tournament.



Figure : Character Class UML Diagram

The attack and defense attributes of the base class will be implemented as arrays of pointers to Die objects in order to accommodate any combination of dice configurations. For example, an attack rating of 2d6 means that the character would roll two six-sided dice to determine the attack force. This would be represented by an array of two Die objects that each return a random integer between 1 and 6. This approach means that we will still need to use an array of pointers for single die; meaning that 1d6, for example, would need an array with a single element. This additional overhead is deemed an acceptable compromise to allow for class inheritance. Since the attack and defense ratings can be different for each derived class, the attack and defense functions are virtual functions. Some of the derived classes also have special abilities that may further modify how the attack and defense functions are implemented. Those differences are outlined in the pseudocode below.

int attack() - // Except Medusa

// Roll each attack die

// Add dice results

// Return the results

int attack() - // Medusa

// Roll each attack die

// Add dice results

// If a 12 is rolled, set results to 100 (Glare)

// Return the results

int defense(int damage) - // Barbarian, Medusa

// Roll each defense die

// Calculate net damage (Damage – armor – defense roll)

// If net damage is less than 0, set net damage to 0

// Decrease strength by net damage

// Return the results of the dice roll

int defense(int damage) - // Vampire

// Generate random number (1 or 2)

// If result is 1

// Roll each defense die

// Calculate net damage (Damage – armor – defense roll)

// If net damage is less than 0, set net damage to 0

// Decrease strength by net damage

// Return the results of the dice roll

// Otherwise, vampire uses Charm and takes no damage

// Return 0 (no need to defend if no attack)

int defense(int damage) - // Blue\_Men

// Roll each defense die

// Calculate net damage (Damage – armor – defense roll)

// If net damage is less than 0, set net damage to 0

// Decrease strength by net damage

// If new strength between 5 and 9, remove one defense die

// If new strength between 1 and 4, remove two defense dice

// Return the results of the dice roll

int defense(int damage) - // Harry\_Potter

// Roll each defense die

// Calculate net damage (Damage – armor – defense roll)

// If net damage is less than 0, set net damage to 0

// Decrease strength by net damage

// If Harry dies and he is on his first life

// Set his strength to 20

// Set second life indicator to true

# Character Class Modifications

A few modifications have been made to the Character class to allow for the additional requirements for tournament play.

First, a new constructor has been defined to allow the user to attach a name to the Character object. This is assigned to the name data member that was also added. The default constructor was also modified slightly to assign a default name to the name data member.

Second, a new member function has been added to provide a method for the fighter to recover some strength after winning a round of combat while waiting to fight again. This function will take as a parameter an integer value representing the amount of damage taken during combat. The pseudocode below described how this function will work.

void recovery(int damage)

// Roll the 1d10 recovery die

// Divide result by 10 to get a percentage in increments of 10

// Multiply the damage by that percentage

// Add that amount back to the fighter’s strength

# Tournament Class

The Tournament class will be implemented simply: two Queue classes to hold each team’s fighter lineup, one DubList class to hold the losers of each battle, two integer data members to hold each team’s score, and an integer to hold the number of initial fighters (which will be the same for both teams). Apart from the constructor and destructor, the class will only have member functions to initialize the teams (including any helper functions for this task), and the fight function (which will control the majority of the battle). Pseudocode for those two functions are shown below.

void initializeTeams(int numFighters)

// For each team, and for the number of fighters specified…

// Display the character choices

// Prompt user for a character choice

// Prompt user for a name

void fight

// Set counter to 1 to count the number of battles (for display)

// Check if either team queue is empty; if not, continue

// Initialize two integers to keep track of cumulative damage

// Get a fighter from each team queue

// Check if the first fighter has strength; if so…

// Set attack and defense roll variables to 0

// First fighter attacks

// Second fighter defends

// Accumulate damage for defender

// Check if defender is dead; if so…

// Send defender to loser pile

// Attacker recovers some strength

// Send attacker to back of team queue

// Assign team points

// Display results of battle

// Display current team queues

// Check if the second fighter has strength; if so, continue

// Set attack and defense roll variables to 0

// Second fighter attacks

// First fighter defends

// Accumulate damage for defender

// Check if defender is dead; if so…

// Send defender to loser pile

// Attacker recovers some strength

// Send attacker to back of team queue

// Assign team points

// Display results of battle

// Display current team queues

# Test Tables

Test tables for previously-created Character class can be found in [Appendix C](#_Appendix_C:_Character). The test table below is only for the newly-added functions to the Character class; namely recovery() and getName(). A test table for the Tournament functions is also provided below.

## Player Member Functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Description/Steps** | **Expected Outcome** | **Observed Outcome** |
| Take two items | Call Player.takeItem() twice  Display inventory | Two inventory items displayed. | Two inventory items displayed |
| Take three items | Call Player.takeItem() three times  Display inventory | Feedback that third item can’t be carried.  Two inventory items displayed. | Feedback that third item can’t be carried.  Two inventory items displayed. |
| Drop an item |  |  |  |

## Space Functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Description/Steps** | **Expected Outcome** | **Observed Outcome** |
| Take an item from the Space | Add items | Validation should catch error and ask again for valid input. | Invalid entry. Please try again. |
| Initialize teams with 1 fighter | Enter 1 fighter when prompted | Prompt for fighter from each team | Prompted for fighter from each team |
| Scoring | Run program with 1 fighter on each team | Winning team should be awarded 2 points. Losing team should lose 1 point. | Winning team score 2.  Losing team score -1 |
| Initialize teams with multiple fighters | Enter 2 fighters per team when prompted.  Choose fighters and names  Run battle | Opponents display.  Winner is displayed.  Winner moves to back of team queue  Loser moves to loser stack | Opponents are displayed.  Winner is displayed.  Winner is moved to back of losing team’s queue  Loser is moved to loser stack. |

# Design Changes

When testing the recovery function, I discovered that the strength was not increasing as expected (or at all, really). In debugging, I discovered that the increase was being rounded down to zero because of integer division, so I needed to change one of my factors to an explicit double.

I also discovered an error where I neglected to change the team for adding the winner when I copied the code from the previous block. This was resulting in the winner, if from team 2, being added to the back of team 1’s queue. Not a design change, but that took me a while to track down. It was also resulting in only a number of battles being fought equal to the initial number of fighters in each queue.

I also discovered that I needed to suppress the output when a special was being used, as it made the output messy.

The last design change I made was to output the contents of each team queue at the end of each battle (after moving winners and losers). I think this helps track that the program is working properly and provides a good visual for the user.

I ran into quite a few memory leak issues. It took some trial and error testing to figure out how to address them. I had it in my head that any dynamic memory allocations need to be freed in the same function or class in which they are initially allocated. This restriction was obviously causing some prob

# Reflection

I ran into quite a few memory leak issues. It took some trial and error testing to figure out how to address them. I had it in my head that any dynamic memory allocations need to be freed in the same function or class in which they are initially allocated. This restriction was obviously causing some challenges to memory management. I finally figured out that it is safe, in the case of pointers to objects as data members for ADT containers, to free the memory for these pointers in the container destructors; even if the memory was initially allocated in client code. This might be something I need to watch for in the future, though, as I can see this tripping me up if I am not cautious using this technique.

Finally, in this project the value of reusing code (and writing code to *be* reusable) has been clearly cemented in my mind. This project would have been much more difficult to complete without most of the classes already having been written as part of earlier assignments. They only needed a little bit of modification to make them work. It’s got me thinking about how to write containers in such a way that they can use inheritance and polymorphism in order to hold any kind of data in their nodes without modifying the original container code (just extending it through inheritance).

# Appendix A: Program Flow Diagram



# Appendix B: Fight Flow Diagram



# Appendix C: Character Class Test Tables

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Description/Steps** | **Expected Outcome** | **Observed Outcome** |
| Barbarian Constructor | Call constructor for Barbarian | No compile error | No compile error |
| getArmor | Call getArmor() and direct output to cout | Display 0 | 0 displayed in console |
| getStrength | Call getStrength() and direct output to cout | Display 12 | 12 displayed in console |
| getType | Call getType() and direct output to cout | Display “Barbarian” | “Barbarian” displayed in console |
| Attack | Call attack() and direct output to cout | Display a random number between 1 and 6 | Compile error. Re-use of attack and defense identifiers. |
| Attack | Call goAttack() and direct output to cout | Display a random number between 1 and 6 | 2 displayed in console |
| Attack (again x3) | Call goAttack() several times and output to cout | Display several random numbers between 1 and 6 | Console displays:  4  37654  37345 |
| Defense and display new strength | Call goDefend(4)  Call getStrength() | Return the actual roll result.  Return value of strength reduced by: 4 – 0 (armor) – X (defense roll result) | Console:  8  16 |
| Vampire charm | Call goDefend(damage) from a Vampire object | Display charm-used message 50% of the time.  Vampire defense roll displays 0.  No damage applied to Vampire | Display charm-used message 50% of the time.  Vampire defense roll displays 0.  No damage applied to Vampire. |
| Blue Men Swarm | Call goDefend() from a Blue Men object with a damage parameter above 15 to trigger defense die loss | Display defense loss, and number of defense dice left. | Display defense loss, and number of defense dice left. |
| Medusa Glare | Set damage to 12 in goAttack() member function for Medusa object. | Display message that glare is used.  Set attack to 100. | Display message that glare is used.  Attack was set to 100. |
| Harry Potter Hogwarts | Set opponent attack to “kill” Harry Potter | Display message that Harry Potter was killed and revived to 20 strength points.  Set Harry’s strength to 20. | Message displayed, but strength not set to 20. |