Shay Snyder

Honors 1260 – Spring 2020

**Zork Design Document**

Problem: 1260 Honors Project 5

**List of Inputs, Outputs, and Processing Required**

Inputs

* Allow the user to control the game by entering various commands
  + - * “go north” – move the player upward
      * “go south” – move the player downward
      * “go east” – move the player to the right
      * “go west” – move the player to the left
    - The user will be able to advance and return to randomly generated cells
    - The user will be able to fight randomly generated monsters
    - The user will be able to escape the dungeon

Outputs

* A visually appealing user interface that display various aspects of the game
* A map of the entire Dungeon
* A display of each room including monsters and weapons
* The player’s health and progress
* A detailed prompt of each fight
* A success screen when the user exits the dungeon

Processing

* Randomly generate a two-dimensional dungeon with each cell potentially containing a monster and weapon
* Use the random class to determine the outcome of a fight between the player and the monster
* Use regular expressions to validate user input
* Determine if the user’s desired move is possible
* Move the user to the next room
* Move the user to the last room
* Decrease the player’s and monster’s health as each fight progresses
* Account the variance of weapons and monsters in each fight
* Determine if the player is still alive
* Allow the player to attack and harm the monster
* Allow the monster to attack and harm the player

**Identification of Classes and Their Responsibilities**

**Class name: ZorkDriver**

Requirements

* Drive the Zork game
* Control all input and output

**Class name: Zork**

Requirements

* Manage all other dependencies of the Zork game
* Hold the global random object
* Display the current status of the game
* Be an intermediary for the user’s desired move

**Class name: Dungeon**

Requirements

* Represent the game’s Dungeon by holding all cells
* Display the dungeon
* Manage the user’s desired move
* Determine the outcome of all fights
* Manage the game’s progression

**Class name: Cell**

Requirements

* Represent each individual cell
* Display the cell’s contents
* Hold a weapon for player pickup
* Determine if a given move is available
* Hold the cell’s monster
* Determine if a cell contains a monster and if so, return said monster

**Class name: Wall**

Requirements

* Represent a Cell wall
* Hold if the wall is an exit
* Hold the wall’s type

**Class name: Monster**

Requirements

* Represent a monster
* Hold the monster’s health
* Hold the monster’s type
* Hold the monster’s damage capability
* Manage attack by the monster
* Determine if monster is alive
* Manage attacks to the monster

**Class name: Skeleton**

Requirements

* Extend the Monster class
* Represent a skeleton
* Set the skeleton’s damage

**Class name: Zombie**

Requirements

* Extend the Monster class
* Represent a zombie
* Set the zombie’s damage

**Class name: Creeper**

Requirements

* Extend the Monster class
* Represent a creeper
* Set the creeper’s damage

**Class name: Player**

Requirements

* Represent a player
* Hold the player’s health
* Hold the player’s type
* Hold the player’s damage capability
* Manage attack by the player
* Determine if player is alive
* Manage attacks to the player

**Class name: Weapon**

Requirements

* Represent a weapon
* Hold the weapon’s damage
* Hold the weapon’s type
* Manage the weapon’s attack
* Set the weapon’s damage to 5

**Class name: Hand**

Requirements

* Extend the Weapon class
* Represent a hand
* Set the weapon’s damage to 3

**Class name: Stick**

Requirements

* Extend the Weapon class
* Represent a stick
* Set the weapon’s damage to 4

**Class name: Sword**

Requirements

* Extend the Weapon class
* Represent a sword
* Set the weapon’s damage to 6

**Class name: Knife**

Requirements

* Extend the Weapon class
* Represent a knife
* Set the weapon’s damage to 5

**Class name: Laser**

Requirements

* Extend the Weapon class
* Represent a laser
* Set the weapon’s damage to 8

**Class name: Gun**

Requirements

* Extend the Weapon class
* Represent a gun
* Set the weapon’s damage to 7

**UML Class Diagram**

A close up of a piece of paper

Description automatically generated

**Algorithms**

**Class:** ZorkDriver

**Method:** main(args : String[]) : void

**Description:** The main method in the ZorkDriver class serves as the backbone for the rest of the program. From this point, the intro screen, main game, and outro screen will be called

**Class:** ZorkDriver

**Method:** intro() : void

**Description:** This method will use JOptionPane to introduce the game to the user.

**Class:** ZorkDriver

**Method:** startgame() : void

**Description:** This method will initialize the actual game and control all interaction between the game and the user via JOptionPane. The user will have four command to control the game: “go east”, “go west”, “go north”, and “go south”. These commands will be used to move the player throughout the dungeon in search for the exit. If a user enters a cell and a monster lies within, a prompt will be displayed detailing various aspects of the fight. If the user enters an invalid command, a prompt will be displayed informing them of the error. Once the user reaches the end of the dungeon, a prompt will be displayed congratulating them on their success.

**Class:** ZorkDriver

**Method:** outro()

**Description:** This method will thank the user for playing the game and wish them a good day.

**Class:** Zork

**Method:** Zork(rnd : Random)

**Description:** This is arg-constructor for the Zork class. Its purpose is to initialize the game by calling the Dungeon(rnd : Random) arg constructor to generate the dungeon

**Class:** Zork

**Method:** display : String

**Description:** this method will be used to inform the user of the game’s current status. This method will return the most relevant data to the user. For example, the string could contain the map of the dungeon.

**Class:** Zork

**Method:** move(move : String)

**Description:**  This method will be called whenever the player decides to make a move. If they make an invalid move, the string will contain “INVALID MOVE! Please try again.” If they simply move to a new cell, the string will contain “SUCCESS! You have advanced to the next cell.” If they move to a previous cell, the string will contain “WARNING! You have returned to the previous cell.” If the user enters a room with a weapon, the string will contain “CONGRATS! You have advanced to the next room and found a new <some weapon>.” If the user enters a room with a monster, the string will contain “You have advanced to the next room and awaken the beast!” and then display various aspects of the fight. If the user exits the dungeon, the string will contain “CONGRATULATIONS! YOU HAVE SUCCESSFULLY ESCAPED”.

**Class:** Dungeon

**Method:** Dungeon(rnd : random)

**Description:** This method will randomly generate a 2D dungeon. The first cell will always be in the left-uppermost cell. From that point, each cell will have an equal likelihood of being placed to the right or below of the previous cell. This algorithm will repeat until the desired number of cells are generated. (I hope to have time to implement a feature where the cells have the potential to generate above the previous cell.)

**Class:** Dungeon

**Method:** showDungeon() : String

**Description:** This method will return a string that contains the layout of the dungeon along with details of each room.

**Class:** Dungeon

**Method:** move(move : String) : boolean

**Description:** This method will handle the user’s desired movement. The method will return true if the move is valid and false if invalid.

**Class:** Dungeon

**Method:** fight(monster : Monster) : String

**Description:** This method is responsible for calculating the result of each encounter between the player and monster. The player has to kill the monster in order to progress through the dungeon.  In every round of the fight, the player will attack first. The player has a 10% chance of missing the match. Every time the player lands a hit, the appropriate amount of damage will be taken from the monster’s health. There is a 20% chance that the monster will miss the player. Every time the monster lands a hit, the player’s health will be reduced by 4 points. This process will be repeated until someone dies. If the player wins, they will be able to advance. If the monster wins, the game is over. The returned string will detail every round of the fight and who won.

**Class:** Dungeon

**Method:** advance() : void

**Description:** This method will increment the currentRoom integer by 1. Thereby, advancing the player to the next room.

**Class:** Dungeon

**Method:** return() : void

**Description:** This method will decrement the currentRoom integer by 1. Thereby, returning the player to the previous room.

**Class:** Cell

**Method:** Cell(walls : ArrayList<Wall>, generateWeapon : boolean)

**Description:** This is the main constructor for the Cell class. The argumented Arraylist is used to build each individual cell. The Wall object at index 0 is the West wall. The Wall object at index 1 is the East wall. The Wall object at index 2 is the North wall. The Wall object at index 3 is the South wall. The argumented boolean specifies whether a weapon will be generated in each room. If the generateWeapon boolean is true, a weapon will be randomly generated.

**Class:** Cell

**Method:** showCell() : String

**Description:** This method will generate a string that contains a visual representation of a given cell and the items within.

**Class:** Cell

**Method:** getHasMonster() : boolean

**Description:** This method returns a boolean variable specifying whether a cell contains a monster

**Class:** Cell

**Method:** getMonster() : Monster

**Description:** This method returns a Monster object containing the monster within a given cell.

**Class:** Cell

**Method:** isMoveAvailable(move : String) : WallType

**Description:** This method returns a WallType value specifying the outcome of the user’s desired move.

**Class:** Wall

**Method:** Wall(type : WallType)

**Description:** This is the argument constructor for the Wall class. The argumented WallType is used to set the Wall’s WallType.

**Class:** Wall()

**Method:** getType() : WallType

**Description:** This getter method returns the WallType of a given Wall object.

**Class:** Skelton

**Method:** Skelton()

**Description:** This constructor is used to initialize the Skeleton class. Since this class extends the Monster class, the constructor sets the Skeleton’s damage to 4.

**Class:** Zombie

**Method:** Zombie()

**Description:** This constructor is used to initialize the Zombie class. Since this class extends the Monster class, the constructor sets the Zombie’s damage to 5.

**Class:** Creeper

**Method:** Creeper()

**Description:** This constructor is used to initialize the Creeper class. Since this class extends the Monster class, the constructor sets the Creeper’s damage to 6.

**Class:** Monster

**Method:** Monster()

**Description:** This constructor is used to initialize the Monster class. Since this class extends the Monster class, the constructor sets the Monster’s damage to 3.

**Class:** Monster

**Method:** attack()

**Description:** This method returns the integer value representing the monster’s damage. This value is used to reduce the health of players.

**Class:** Monster

**Method:** decreaseHealth(damage : int) : void

**Description:**  This method reduces the monster’s health by the argumented integer.

**Class:** Monster

**Method:** getIsAlive() : Boolean

**Description:**  This method returns a Boolean value specifying if a monster is still alive.

**Class:** Player

**Method:** Player()

**Description:** This is the default constructor for the Player class. The player’s health attribute is set to 100, The alive Boolean attribute is set to true. The weapon attributed is set to hand.

**Class:** Player

**Method:** getHealth() : int

**Description:** This method returns the health count of the player.

**Class:** Player

**Method:** getIsAlive()

**Description:** This method returns the living status of the player.

**Class:** Player

**Method:** getWeapon() : Weapon

**Description:** This method returns the weapon currently held by the player.

**Class:** Player

**Method:** decreaseHealth(damage : int) : void

**Description:** This method reduces the player’s health by the argued integer. If their health is less than or equal to zero, set the alive Boolean to false;

**Class:** Player

**Method:** attack() : int

**Description:** This method returns an integer value that represents the damage dealt by a player’s attack.

**Class:** Weapon

**Method:** Weapon()

**Description:** This is the default constructor for the Weapon class. The damage attribute is set to 5. The type attribute is set to “General”.

**Class:** Weapon

**Method:** attack()

**Description:** This method returns the  integer value representing the weapon’s damage.

**Class:** Hand

**Method:** Hand()

**Description:** This is the default constructor for the Hand class which extends the Weapon class. The damage attribute is set to 3. The type attribute is set to “Hand”.

**Class:** Stick

**Method:** Stick()

**Description:** This is the default constructor for the Stick class which extends the Weapon class. The damage attribute is set to 4. The type attribute is set to “Stick”.

**Class:** Sword

**Method:** Sword()

**Description:** This is the default constructor for the Sword class which extends the Weapon class. The damage attribute is set to 6. The type attribute is set to “Sword”.

**Class:** Knife

**Method:** Knife()

**Description:** This is the default constructor for the Knife class which extends the Weapon class. The damage attribute is set to 5. The type attribute is set to “Knife”.

**Class:** Laser

**Method:** Laser()

**Description:** This is the default constructor for the Laser class which extends the Weapon class. The damage attribute is set to 8. The type attribute is set to “Laser”.

**Class:** Gun

**Method:** Gun()

**Description:** This is the default constructor for the Gun class which extends the Weapon class. The damage is set to 7. The type attribute is set to “Gun”.

**Test Cases**

These are the test cases that were used to test our project. The various inputs and outputs are detailed below.

**Test Case #1**

 Input: the user starts the game

Output: the into screen should be displayed

**Test Case #2**

Input: the user types “GO EAST”, “GO NORTH”, ”GO WEST”, “GO SOUTH”

Output: attempt move the player to the right, north, west, or south

**Test Case #3**

Input: the user types “go east” , “go north”, “go west”, “go south”

Output: attempt to move the player to the right, north, west, or south

**Test Case #4**

Input: the user types “east”, “north”, “west”, “south”

Output: attempt to move the player to the right, north, west, or south

**Test Case #5**

Input: the user types anything but the aforementioned inputs

Output: display a prompt informing the user of the error

**Test Case #6**

Input: The user exits the dungeon

Output: Congratulate the player for completing the game

**Test Case #7**

Input: The user prematurely terminates the program

Output: Ask the user if they are sure they would like to exit the program.

**Test Case #8**

Input: The user dies during a fight

Output: Ask the user if they would like to play again.

**Test Case #9**

Input: The user kills the monster

Output: The details of the fight will be displayed to the user and the game will proceed.