Software Design Description

Version 1
03/22/2021

"Around the World”
Object Design

Ashley M, Kris H, Ryan G

Table of Contents

List of Figures

1.0. Introduction

1.1. Purpose

The is the Software Design for “Around the World” text adventure game. This document will outline the software design and specification of our workflow. In addition to system architecture, system components, and software requirements as described in the software requirements.

1.2. Scope

“Around the World” is a game that will allow a user to move from one city to another. In some cities the user will interact with puzzles and monsters. Other cities will contain items that the user could add or remove from their inventory. These items will be useful when interacting with a monster in a specific city. The game will display a message containing the users final score once they have completed going through each city.

“Around the World” will be designed to play on the device of a PC. The game will be written in Java programming language and will text based. The game will also work with a SQLite database that will contain the rooms, monsters, puzzles, and items.

The system will have three packages that interact with each other. These packages will have the components of a Model View Controller architecture. Each package will contain objects with their own attributes that establish what the game will look like. The subsystems will transfer data to one another so the user can use the commands provided.

1.3. Glossary

Attribute – a specific characteristic or value of the object.
Class – creates an object that contains values and methods
HP – player's health points
Methods – performs a specific procedure inside an object class
PC – personal computer
SQLite database – an application development that stores attributes.

Subsystem/Package – provides a service for a system.
System – set of objects that interact with each other

1.4. References

Team 4's Requirements Analysis: Team2's Around the World .docx

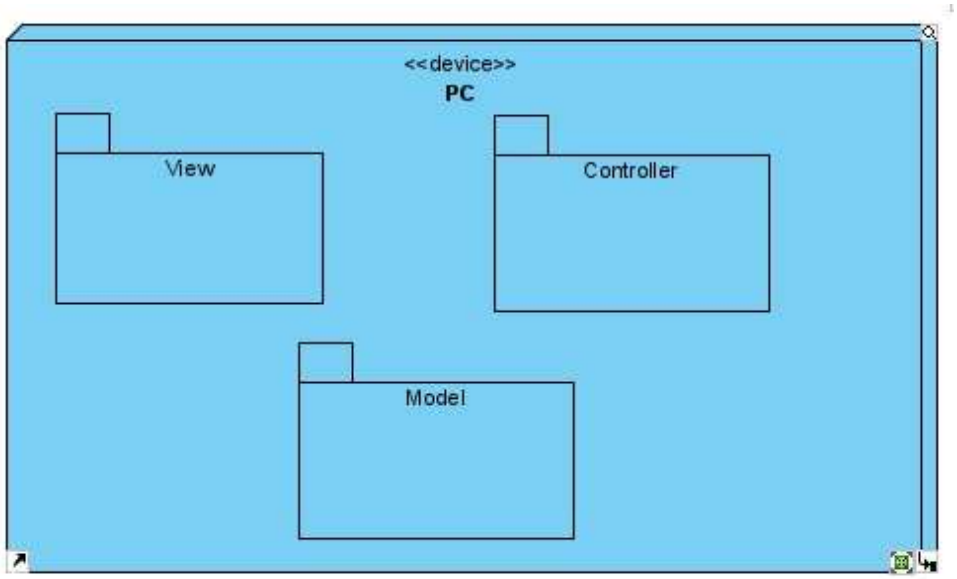
Team 2's Around the World Requirements Document .docx

1.5. Overview of Document

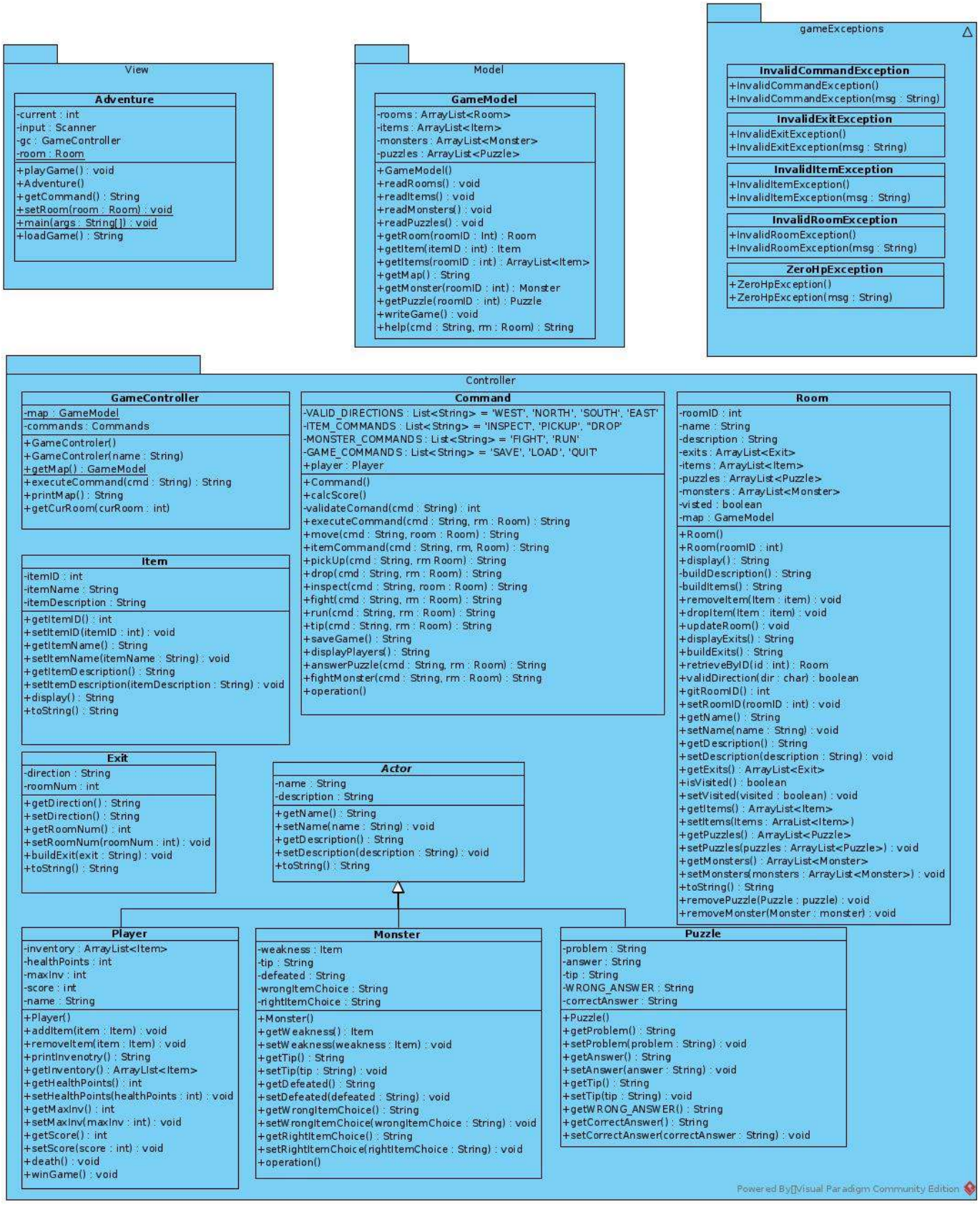
This system design document contains information about how the system will be deployed in a deployment diagram, architecture of the system and how classes will interact in a class diagram, the structure of the database including tables, use case realizations that overview interactions between the user and system, as well as interactions between the different objects within the system, a brief overview of the UI, and an explanation of the help system.

2.0. Deployment Diagram

"Around the World" will be played on the device PC. The PC device will consist of three subsystems. View, Controller and Model. The subsystem View will oversee the user interaction with the game which will call methods from the subsystem Controller. The subsystem Controller will contain methods that will allow capabilities for the user interface. The Controller will have objects each containing their own methods on how they will be interpreted. The subsystem Model will respond to the methods from the subsystem Controller to retrieve the objects to create a map of the game which the subsystem View will also have access of.



3.0. Architectural Design



Adventure class which will be in the View package will contain the current location of the player and will call the Game Controller class from the Controller package to retrieve the current map and execute the commands of the user input.
Game Model class is in the Model package will contain the whole map of the game. It will read the rooms and items and get the monster and puzzles for the designated rooms.
The game Exception package will contain classes of exceptions to handle incorrect command input, exit input, room input and item input.
Game Controller class will be inside the Controller package. This class will contain an object of Game Model class and Command class. With these objects the Game Controller class can retrieve the map and commands to execute the correct command in the current room.

The Command class will be in the Controller package and will contain the attributes of different types of commands. There will be a list of valid directions, a list of valid item interactive commands, a list of valid monster interactive commands, and general game commands such as save, load, or quit. The class will contain a Player object so that it can calculate the score and update the players inventory list. The user input will be validated inside the Command class and executed. Each command will have their own method that will implement a text for the user.

Room class will have the attributes that make up a room such as the ID, name, description. There will be a list of exits, monsters, puzzles that can be retrieved and modified. The class will update the room whenever the player has visited or has interacted with an item, puzzle, or monster.

The Exit class will contain the exits for each room and get the description of the correct room associated with the exit.

Item class will have attributes that make up an item such as the ID, name, description and can be retrieved and altered.

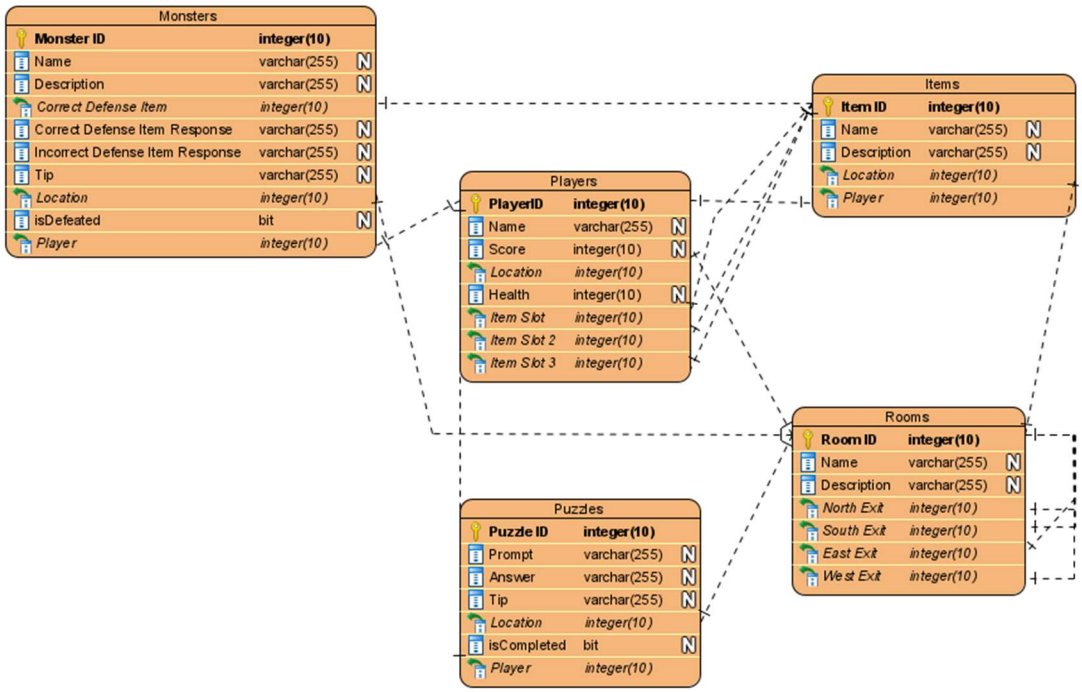
The Actor class will be abstract meaning it will not contain an actual object of its own instead it will inherit methods from the Player, Monster, and Puzzle class. These subclasses will be easily accessed through the Actor class by calling the methods to the Command class as well as the Game Controller class.

The Player class will contain the names of the users and a list of inventories which can only be up to three items. The inventory can be added in to or remove items. The health attribute will start the user at 100 and decrease each time an item for defense in Monster class is used incorrectly or puzzle answer is incorrect. The score will be determined inside the Command class. The attributes of the Player class will all be retrieved whenever the user inputs their text that is taken place in the Adventure class which will then head to the Command class to validate the command and display the appropriate message. This will be updated in the Game Controller class which will also update the Actors objects.

In the Monster class it will have attributes of the item to use to defeat the monster and a tip to help the user. A monster will be in several rooms and depending on the monster selected the User must input the right text command to defeat the monster. The user input will be validated in the Command class and display the appropriate message to the user. The user will then retrieve any items from the Inventory class that the user has already added into their inventory. The inventory will be updated each time the Player object adds or removes an item from the list of inventories.

Puzzle class will contain attributes such as prompt, answer, and TIP for each puzzle listed. Each puzzle will be answered by receiving user input from the Adventure class main method. The main method will call the Command class to validate and display the correct message to user. The user will then respond and repeat the process. During all this the Game Controller class will be updating the map to ensure the Player object is in the right spot. Once the puzzle is answered correctly the Game Controller will update the rooms and allow the Player object to move to the next room. If answer is not correct to the puzzle, then the Player object will not have any commands available to move.

4.0. Data Structure Design



The above design diagrams a relational database consisting of 5 tables to be used as Around the World's backend. Utilizing this specific structure, the game has a central store that I can use to both store and load the game and multiple player profiles.

Tables Reference

Rooms

The Rooms table contains entries denoting the specifics of all rooms to be used in the map.

Fields

RoomID – An ID used to denote a specific room.
Description – Room's description to be displayed upon entering.
North Exit – An ID used to denote the room that can be accessed by going through the north exit.
South Exit – An ID used to denote the room that can be accessed by going through the south exit.
East Exit – An ID used to denote the room that can be accessed by going through the east exit.
West Exit - An ID used to denote the room that can be accessed by going through the west exit.

Foreign Keys

All Exits (North, South, etc.) serve as foreign keys that are linked to the primary key (RoomID) of the current Rooms table.

Monsters

The Monsters table contains entries denoting the specifics of all monsters to be included in the game.

Fields

MonsterID- A specific ID used to denote a specific monster.
Name- The monster's name to be displayed.
Description- The monster's description to be displayed.
Correct Defense Item- The ID used to denote the specific item used to defeat the respective monster.
Correct Defense Item Response- The printed response when the Player selects the correct item to use against the monster.
Incorrect Defense Item Response- The printed response when the Player selects the incorrect item to use against the monster.
Tip- A response containing some helpful information that the Player can use when they encounter a specific monster.
Location- The ID used to denote the specific room in which the respective monster is placed.
isDefeated- A true/false value indicating whether the monster has been defeated.
Player- A specific ID used to denote a Player.

Foreign Keys

Correct Defense Item is a foreign key that links to the primary key (ItemID) of the Items table.
Location is a foreign key that links to the primary key (RoomID) of the Rooms table.
Player is a foreign key that links to the primary key (PlayerID) of the Players table.

****Note: The Monsters table contains a Player foreign key for the sole purpose of keep tracking of whether that player has defeated a certain monster. While this might add more entries to the**

respective table, it ensures that we could keep track of certain monsters and show/hide them for certain players, as necessary. **

Puzzles

The Puzzles table contains entries denoting the specifics of all puzzles to be included in the game.

Fields

PuzzleID- A specific ID used to denote a specific puzzle.
Prompt- The prompt serves to contain the actual puzzle.
Answer- The answer for the puzzle.
Tip- A response containing some helpful information that the Player can use when they encounter a specific puzzle.
Location- The ID used to denote the specific room in which the respective puzzle is placed.
isCompleted- A true/false value indicating whether the puzzle has been completed.
Player- A specific ID used to denote a Player.

Foreign Keys

Location is a foreign key that links to the primary key (RoomID) of the Rooms table.
Player is a foreign key that links to the primary key (PlayerID) of the Players table.

****Note: Like the Monsters table, the Puzzles table contains a Player foreign key for the sole purpose of keep tracking of whether that player has completed a certain puzzle. While this might add more entries to the respective table, it ensures that we could keep track of certain puzzles and show/hide them for new/returning players, as necessary. ****

Items

The Items table contains entries denoting the specifics of all items to be included in the game.

Fields

ItemID- A specific ID used to denote a specific item.
Name- The item's name to be displayed.
Description- The item's description to be displayed.
Location- The ID used to denote the specific room in which the respective monster is placed.
Player- A specific ID used to denote a Player.

Foreign Keys

Location is a foreign key that links to the primary key (RoomID) of the Rooms table.
Player is a foreign key that links to the primary key (PlayerID) of the Players table.

****Note: The Items table contains a listing of all items included in the game, both in rooms and in the inventory. In the case of specific rooms, the Location key is linked to a specific RoomID. In the case of inventory items, the Location Key is nullified in the Items table and a new reference is created from the Players table. ****

Players

The Players table contains entries denoting all players, their inventories, and specific player profile information.

Fields

MonsterID- A specific ID used to denote a specific player.
Name- The player's name. This is the attribute used to separate and identify different players and profiles.
Score- The player's game score.
Location- The ID used to denote the specific room in which the player is currently standing.
Health- The player's health (from 1-100).
Item Slot- The ID used to denote a specific item that the player has placed in their Inventory's first slot.
Item Slot 2- The ID used to denote a specific item that the player has placed in their Inventory's second slot.
Item Slot 3- The ID used to denote a specific item that the player has placed in their Inventory's third slot.

Foreign Keys

Location is a foreign key that links to the primary key (RoomID) of the Rooms table.
Item Slot fields are foreign keys that link to the primary key (ItemID) of the Items table.

5.0 Use Case Realizations

1. Load Game

User loads game	1. Main starts the game, initializes a Scanner, a GameControler and calls playGame() 2. PlayGame() prints the intro to the game and "LOAD game or NEW game"
User inputs "LOAD <name of player>"	1. loadGame() takes the <name of player> and checks if player exists 2. LoadGame creates an instance of gameControler class 3. GameController() will load the game using model to readRooms(), readItems(), readMonsters(), and readPuzzles()

2. Save Game

1. User inputs "SAVE"	1. GetCommand sends the string to game controler 2. GameControler calls validateCommand() 3. ValidateCommand() parses the string, checks if it is a valid command, and calls saveGame()

	4. SaveGame calls the gameModel writeGame() to save current game in database 5. Returns “Save game <playerName>” to getCommand

3. New Game

1. User loads game	1. Main starts the game, initializes a Scanner, a GameControler and calls playGame() 2. GameControler() instantiates a new GameModel and Commands object and calls GameModel readItems, readRooms, readMonsters, and readPuzzles to build the game 3. PlayGame() prints the intro to the game and “LOAD game or NEW game”
2. User inputs “NEW”	1. PlayGame() starts the game

4. Quit

1. User inputs “QUIT”	1. GetCommand sends the command to the gameControler 2. GameControler calls validateCommand to check command is valid 3. System exits

5. Help

1. User inputs “HELP”	1. GetCommand sends the command to the gameControler 2. GameControler calls validateCommand to check command is valid 3. GameContoler calls help method from game model

	4. Help reads string from HELP.txt and returns the string to the gameControler 5. GameControler displays help string to the user

6. Drop Item

1. User inputs “DROP”	1. GetCommand sends the string to the gameController 2. GameController calls the validate command to command is valid and sends reponse string “What item would you like me to drop?”
3. User inputs “<item name>”	1. GameControler sends item string to drop method 2. Drop method finds the item object and updates the playerinventory by calling removeitem from player class, and room by calling dropitem from room class 3. Drop method returns reponse string back to gameController “You have dropped <item name>”

7. Pick up item (player has inventory space)

1. User inputs “PICK UP”	1. GetCommand sends the string to the gameControler 2. GameController calls validateCommand on string command 3. ValidateCommand checks to see if the command is valid and send the command to pickUp 4. GameControler checks if the player has less than 3 items, if false sends string “What item would you like to pick up?”
2. User inputs “<item to be picked up>”	1. GameController calls pickUp method 2. PickUp finds the correct item object and updates player inventory by calling addItem from

	player class, and room by calling removeItem from room class. 3. Pickup method return response string back to gameController "You have picked up <item to be picked up>"

8. Pick up item (player does not have inventory space & inputs "NO")

1. User inputs "PICK UP"	1. GetCommand sends the string to the gameControler 2. GameController calls validateCommand on string command 3. ValidateCommand checks to see if the command is valid and send the command to pickUp 4. GameControler checks if the player has less than 3 items, if false sends string "You have 3 items on you already, would you like to switch with an item from your inventory? YES or NO?"
2. User inputs "NO"	1. GameControler responds with "You decided to keep your 3 items"

9. Pick up item (player does not have inventory space & inputs "YES")

1. User inputs "PICK UP)	1. GetCommand sends the string to the gameControler 2. GameController calls validateCommand on string command 3. ValidateCommand checks to see if the command is valid 4. GameControler checks if the player has less than 3 items, if false sends string "You have 3 items on you already, would you like to switch with an item from your inventory? YES or NO?"

2. User inputs “YES”	1. GameController responds with “What item would you like to swap?”
2. User inputs “<item to swap>”	1. GameControler sends item string to drop method 2. Drop method finds the item object and updates the playerinventory by calling removeitem from player class, and room by calling dropitem from room class 3. Drop method returns reponse string back to gameController “You have dropped <item name>” 4. GameControler checks if the player has less than 3 items, if false sends string “What item would you like to pick up?”
3. User inputs “<item to be picked up>”	4. GameController calls pickUp method 5. PickUp finds the correct item object and updates player inventory by calling addItem from player class, and room by calling removeItem from room class. 6. Pickup method return response string back to gameController “You have picked up <item to be picked up>”

10. Move

1. User inputs “MOVE <direction>”	1. GetCommand sends the string to the gameController 2. GameController calls validateCommand on string 3. ValidateCommand parses the command and checks if it is valid and if the direction exits and calls the move method 4. Move method returns the string for the new room, updates the exited room to visted=true, and updates in the currentRoom in Adventure class 5. Checks for a puzzle, if false...

	6. Checks for a monster, if false... 7. Checks for items and creates a string of items 8. Returns the string of items to the gameControler 9. GameControler prints items to user.

11. Move encounter puzzle

1. User inputs “MOVE <direction>”	1. GetCommand sends the string to the gameController 2. GameController calls validateCommand on string 3. ValidateCommand parses the command and checks if it is valid and if the direction exits and calls the move method 4. Move method returns the string for the new room, updates the exited room to visted=true, and updates in the currentRoom in Adventure class 5. Checks for a puzzle, if true.. 6. GameControler initiates Encounter Puzzle 7. Checks for a monster, if false... 8. Checks for items and creates a string of items 9. Returns the string of items to the gameControler 10. GameControler prints items to user.

12. Move encounter monster

1. User inputs “MOVE <direction>”	1. GetCommand sends the string to the gameController 2. GameController calls validateCommand on string 3. ValidateCommand parses the command and checks if it is valid and if the direction exits and calls the move method

	4. Move method returns the string for the new room, updates the exited room to visited=true, and updates in the currentRoom in Adventure class 5. Checks for a puzzle, if false... 6. Checks for a monster, if true... 7. GameControler initiates Encounter Monster 8. Checks for items and creates a string of items 9. Returns the string of items to the gameControler 10. GameControler prints items to user.

13. Encounter Puzzle correct answer

	1. GameControler sends puzzle string to user
1. User inputs answer	1. GameControler compares the user's input with answer string with answerPuzzle method 2. If they are equal answerPuzzle() removes the puzzle from the room with removePuzzle method from room class, updates player score +10 by calling setScore pulls correctAnswer string and print it to the user.

14. Encounter Puzzle incorrect answer

	1. GameControler sends puzzle string to user
1. User inputs answer	1. GameControler compares the user's input with answer string. 2. If they are not equal the answerPuzzle() method will send "The answer to the question is incorrect, all of a sudden you feel your life drain a bit. You lost 5 health points.", will update player hp by calling setHealthPoints

	method on player, checks if hp is above zero and if false throws a ZeroHpException 3. Resumes at step 1

15. Encounter Monster correct item

	1. GameControler sends string “Monster found", monster name, and “FIGHT or RUN” to user
1. User inputs “FIGHT”	1. GameControler sends command to validateCommand method which parses and calls the fight() method. 2. The fight method sends “What item will you use?” to the user
2. User inputs <item name>	1. fight method takes <item name> and finds the correct item object to match the string and compares it to monster weakness with getWeakness() method. 2. An InvalidItemException is thrown if the user doesn’t have the item 3. FightMonster method sends correct item string and “Congratulations you defeated <monster name>. You are able to move to the next room.”, removes the monster from the room with removeMonster() method, removes the item from the player’s inventory with removeItem() method and updates the player’s score +10 with setScore() method.

16. Encounter Monster incorrect item

	2. GameControler sends string “Monster found", monster name, and “FIGHT or RUN” to user
2. User inputs “FIGHT”	3. GameControler sends command to validateCommand method which parses and calls the fight() method. 4. The fight method sends “What item will you use?” to the user

3. User inputs <item name>	4. fight method takes <item name> and finds the correct item object to match the string and compares it to monster weakness with getWeakness() method. 5. An InvalidItemException is thrown if the user doesn't have the item 6. FightMonster method() updates player HP –10, sends string for incorrect item with getWrongItemChoice() method 7. Throws ZeroHpException is player hp drops to zero 8. Resumes back at step 1

17. Encounter Monster RUN

	3. GameControler sends string "Monster found", monster name, and "FIGHT or RUN" to user
3. User inputs "RUN"	5. GameController sends command to validateCommand method which parses and calls the run() method. 6. The run method changes the player's current room to the previous room with Adventure.setRoom()

18. Get Tip

1. User inputs "TIP"	1. GetCommand() sends the string to the gameController 2. GameController sends command to validatesCommand() which parses and calls tip() method 3. Tip() method checks for a monster or puzzle object in the room. 4. If there is not monster or puzzle returns "Sorry there is no tip available."

	5. If there is a monster or puzzle tip() will pull the correct tip string and send it to the game controller 6. GameControler will print the tip to the user

6.0 User Interface Design

The UI will be the console. Text output should be organized and easy to read by using spaces, new lines, and punctuation that well guide the user's eyes during the game.

7.0 Help System Design

The Help System will be accessible to the user through both the HELPME.txt and by typing the HELP command from the console. The HELP command will display the HELPME.txt.

Index