The Oregon Trail with Python(s)

M04 Design Document

Red Team

SDEV265 Summer 2023

Contents

[Introduction 3](#_heading=h.26in1rg)

[UML Diagram 4](#_heading=h.lnxbz9)

[Architecture Design Diagram 6](#_heading=h.35nkun2)

[Class Diagram 8](#_heading=h.1ksv4uv)

[Entity Relationship Diagram 10](#_heading=h.44sinio)

[Interface Design 11](#_heading=h.2jxsxqh)

[Use Cases 12](#_heading=h.z337ya)

[Test Cases 14](#_heading=h.3j2qqm3)

[Database Design 17](#_heading=h.1y810tw)

[Summary 17](#_heading=h.4i7ojhp)

# Introduction

The purpose of our project, titled The Oregon Trail with Python(s), is to recreate the popular text-based adventure decision game from the ‘70s “The Oregon Trail” using python. This will not be a 1:1 recreation but simply inspired by it. This version will have python puns included in gameplay to reference the language the game is built with, Python. The software that this team will use is an operating system (preferably windows), Python, and Visual Studio Code. The hardware requirements for this project include a Windows computer, a monitor, and peripherals such as a keyboard and mouse.

The purpose of this phase is to begin to build the structure of our program. We are outlining how the game will be coded and how all of the pieces will work together to make a functioning program. We’re also providing use cases and test cases for the game. The following report will discuss the UML diagram, architecture design diagram, class diagram, the entity relationship diagram, interface design, use cases, test cases, and database design.

# UML Diagram

The following is a unified modeling language diagram for how the user or actor interacts with the game. The actor will first choose a character. That will update the player class with the desired character and their attributes. Then, the player class will request a scenario where the player can buy food. The scenario requests an action from the actor to choose the amount of food that they want to purchase. This updates the player food and player money in the player class. This will be done in many different scenarios. Not all of them will updates money or food but will present new options and events to the actor. As the actor is presented with new scenarios, they travel through the checkpoints which are presented by the checkpoint manager. As these are presented, the distance is updated for each scenario. Eventually, the player will make their way through each checkpoint and arrive at the final destination or die along the way either by scenario or by running out of food.



# Architecture Design Diagram

Below is the architecture design diagram we plan to use for The Oregon Trail with Python(s). This team is using a layered architecture approach. We have broken the development up into four independent layers. First, there is the existing layer that we plan to use as our base, the operating system. This could be Windows, Mac, or Linux. It would also include a Python installation. Second, is the core logic and application functionality for the game. This would be our game engine which manages the interactions between all of the pieces. It includes our mileage counter, our food counter, our money counter, and more. Next, we have the user interface management, a layer of scenarios that the user will interact with through a GUI. The scenarios will connect the game engine and the GUI together and provide the play options. Finally, the top layer is the user interface itself. This is what the user will actually see and physically interact with. These will be visual representations of the coded scenarios and game engine statuses which will be compiled using a TKinter library. Together, it will form the architecture for our game.



# Class Diagram

The following is a class diagram for the inner workings of the game. The player is our main class that ties together all of the other functionalities. This class contains the characteristics of the player. It also contains the current status of the player’s food and money. Hunting and travel will affect the player status information. The scenario manager automatically loads each scenario from their respective file and makes them accessible to the rest of the code in the program. This is where the user will be presented with various events and challenges. Some will require the user to make a choice as to how they would like to proceed. These scenarios will affect the player class by adding or removing food, adding or removing money, doing nothing, adding or removing distance to the next checkpoint, or killing the player. The checkpoint manager class controls the flow of the game, dictates when a player arrives at a city or travels, or when the game is over.

A picture containing text, diagram, screenshot, parallel

Description automatically generated

# Entity Relationship Diagram

The below is an entity relationship diagram that describes how the different entities in our game are interrelated. The user selects a character from three options (farmer, banker, and merchant). Each character has different skills which affect the outcomes of the scenarios they are faced with. The scenarios then affect the food, money, and distance the player has. Eventually, after running through the scenarios, the player gets to their destination or dies along the way.

A picture containing diagram, sketch, technical drawing, plan

Description automatically generated

# Interface Design

The following is the design of the interface. First, the player begins on a start page. They select their character and then make their initial purchases. Next, the player chooses if they would like to travel or stay and hunt. This will display different scenarios for them and lead them to the next checkpoint. The player will either die on the trail and end the game, leading to a “you died” screen or they will reach their final destination and receive a “You arrived” screen.

A picture containing diagram, sketch, technical drawing, plan

Description automatically generated

# Use Cases

The follow is the diagram of uses cases for the user. You can see how the player will interact with the various components of the game and how those choices will in turn affect the player. The player will make a character selection. This will affect the scenario outcomes that they are presented with. As the player travels, they will be presented with many different scenarios. The scenario will present itself to make a decision, and these decision options will be presented to the player. The player will make their selection and their decision will be relayed back to the scenario. This will put them closer to the next checkpoint (or delay their progress/kill them). After reaching the next checkpoint, the player will have the option to buy food. This will be presented to the player, and it will update the player stats. Additionally, the player can choose to hunt instead of traveling during a day. This will ideally add food to their player stats, unless they are not successful in hunting.



# Test Cases

The following table shows the different test cases that we will need to examine during the testing phase of our project.

| Test case ID | Test case description | Test steps | Expected result |
| --- | --- | --- | --- |
| TC\_StartScreen\_01 | After opening the game, the game should start, display the option to begin the journey, or exit the game and ask the user for input. | * Navigate to the game URL. * Click on start. | Games begin flawlessly and present the user with the preferred options to choose from, along with the choice of user input. |
| TC\_StartScreen\_02 | Test valid selection to make sure an error message pops up. | * Choose an incorrect choice from the given options in the user entry field. | An Error message is displayed. |
| TC\_StartScreen\_03 | The game should not proceed until proper input has been entered. | * Continue picking invalid options. * Enter a valid option | * Error message keeps displaying. * Move to the next scene. |
| TC\_Character | To see if the user gets what comes with a character after selecting one from the available choices. | * Choose the “Banker” character. | * After selecting the “Banker” character, the game starts with $2000 money and no useful skills or wilderness experience in special scenarios. |
| * Choose the “Merchant” character. | * After choosing the "Merchant" character, the player starts the game with $1200, receives a 5–10% discount on purchases, and gets average skills and wilderness experience in some scenarios. |
| * Choose the “Farmer” character. | * When the "Farmer" character is chosen, the game begins with $700 and grants players access to knowledge of plants and wildlife in specific situations. |
| TC\_Money | Verify if the user can purchase food or a ferry. | * Choose the option that takes you to the store. * Checkout or exit the store. | Goods purchased should be added to the character's account, and the cost of those purchases should be subtracted from the total amount of money the character possesses. |
| TC\_Time | Check if the distance decreases and if the counter at the backend keeps track of the distance covered. | * Travel from one checkpoint to the other with a character. | After traveling between checkpoints, distance should decrease at random and within a certain range with each day spent traveling. |
| TC\_Food | To test that the food counter is in good working order and that the amount of food displayed is accurate. | * Purchase food at the store. | Food purchased before the journey should be displayed accurately. |
|  | Check if the food counter tracks the amount consumed each day and reduces the amount left. | * Travel from one checkpoint to the other with a character. | The amount of food consumed while traveling should be tracked and deducted from the total amount of food. |
|  | Check if the user is allowed to purchase food at certain checkpoints | * Travel from one checkpoint to the other with a character. * Verify whether purchasing food is an option. * Purchase more food. | Food purchased should be added to the amount of food left. |
| TC\_Checkpoint | At each checkpoint, check if the users have the option to buy food, and if the merchant receives a random discount. | * Select the “Merchant” character. * Travel from one checkpoint to the other. * Verify whether purchasing food is an option. * Purchase more food. | * The merchant receives a random 5–10% discount on food units at certain checkpoints, rounded up to the next whole dollar. |
| Check if there are rivers between checkpoints. | * Travel from one checkpoint to the other. | * There should be a predetermined number of rivers to cross between checkpoints. It will resemble a checkpoint but won't have a food market. |

# Database Design

Our project does not have a need for a database. Much of our data needs to be loaded from the file system or from code to be efficient with our architecture as a whole.  Because there are no tracked stats, no data to save, and no problems that a database is needed to solve, implementing a database as part of our project as it is would add unnecessary complexity for minimal value. It is difficult to include a database as a necessary and efficient choice with our project as it is now.

Depending on if the project is finished ahead of schedule, there may be an opportunity to make a database of high scores. Should this happen, other alternative methods of saving high scores, such as saving to a text file, would need to be evaluated. Incorporating a database system means we would need to expand our software toolchain and might introduce new errors or decrease the robustness of our project. Going with a simple text file and parser might provide the same value for less development time and more simplicity.

# Summary

The Oregon Trail is a historically significant video game, and while our adaptation is not a 1:1 recreation, it will be an entertaining tribute to the original with Python puns. This design document contains the UML diagram, architecture design diagram, class diagram, the entity relationship diagram, interface design, use cases, test cases, database design, and other information used to plan for and ensure that our recreation is feasible and robust.