Project Title: Small Farm Game

Use Case Description

The Small Farm Game is an interactive and educational simulation game designed to provide users with experience in managing a small-scale farm. The target users of this project are players who enjoy casual simulation games, and are looking for a simple, engaging game in which they manage virtual resources, grow crops, and raise animals. Users are in control of a character who can move around a farm, plant and cultivate crops, raise animals, water plants, feed livestock, and harvest or sell farm products.

The main purpose of this project is to apply Object-Oriented programming principles to develop a functional and enjoyable farming simulation game for users to play. The game's design will encourage strategic planning as players try to expand their farms while managing their resources, money, and limited time. The core gameplay loop will involve the players planting seeds, feeding animals, monitoring growth, harvesting, and making decisions about purchasing, and selling resources. The aim is to deliver a functional, interactive simulation-based game which entertains the user.

Class List and Description:

Item (Base Class):

The item base class is the parent class for all objects that can be bought, sold, or managed within the farm, including harvested crops and raised animals. It will include fundamental key attributes such as the selling price, buying price, and quantity for items, atop of having virtual functions for methods to buy and sell said objects. Notably, entities like plants and animals will be converted into Item objects upon harvesting or slaughter, enabling them to be sold or reused. By abstracting these common properties, the item class enables polymorphism for handling common item behaviours and allows different item types to be treated uniformly in inventory and economy systems.

Entity (Base Class for farm entities):

Entity will be a base class that acts as the parent for all objects that can be placed in the game world, such as plants or animals. It will include pure virtual functions which must be defined in child classes to define behaviours such as time progression, growth progression, growth stage, and other common methods. Entity classes are not directly buyable or sellable but rather provide methods for tracking the life cycle for game-world entities like animals and plants.

Plant (Inherits from Entity):

The Plant class is the parent class for all crop-based entities in the game and handles the lifecycle of plants from seeding, to growing and harvesting. It defines and expands on the behaviours provided by the Entity class and contains attributes such as the position. The Plant class essentially tracks and controls the plant's growth progress, which depends on if it has been watered. Upon reaching maturity, the plant becomes harvestable and can be harvested. The harvest method will remove the plant from the plot as

an entity and return a corresponding Item that represents the yield of the crop into the player's inventory, where it can be sold or reused. This will set as a pure virtual function to ensure implementation in children classes.

CarrotPlant (Inherits from Plant):

The CarrotPlant class represents a carrot entity planted in the game world that grows over a fixed number of time steps when watered. It inherits most of its functionality from the Plant class, including positioning and time progression. The CarrotPlant only progresses through its growth stages when watered before each time step. Once fully mature, it becomes harvestable and deposits an according number of carrots into the player's inventory.

PotatoPlant (Inherits from Plant):

The PotatoPlant class is identical to the CarrotPlant class with the main major difference being the visual growth stages, and the time required to grow. Upon harvesting, it deposits an according number of potatoes into the player's inventory.

Animal (Inherits from Entity):

Analogous to the Plant class, the Animal class manages the life cycle and evolution of livestock. It expands on the behaviours provided by the entity class and contains attributes such as the animal's growth stage, feeding status, and slaughtering status. The Animal class tracks and controls the animals progress which depends on if they have been fed. Once an animal is fully raised, it can be slaughtered, which returns a corresponding item into the player's inventory, or used for reproduction (this is a future feature and is not planned for the initial release but may be implemented if time permits).

Cow (Inherits from Animal):

The Cow class represents a cow entity being raised in the game world. The cow grows and evolves over a set number of days when fed, before reaching maturity at which point it can be slaughtered for items or used for reproduction. Upon slaughtering, the cow deposits an according number of beef into the players inventory.

Player:

The Player class represents the user character, and tracks their money, position and inventory. Players can interact with the world by watering plants, feeding animals, planting crops, harvesting crops, slaughtering animals, and selling/buying goods. When harvesting or slaughtering, the player obtains an item corresponding to that action, which is added to the player's inventory.

Inventory:

The Inventory class will be responsible for storing and managing all Items owned by the player. It will essentially be a map relating each item type to the quantity possessed by the player and will also handle features such as adding new items, removing items, and getting item counts. The inventory is essentially an index of all items owned by the player and will be used to bridge the gap between items and entities;

Items will be added when entities are harvested, while entities will be placed in the game world when items are used.

Plot:

The plot is a general-purpose container for game world entities. It provides spatial management, which entails (x,y) coordinates of its top left corner, and dimensions of length and height. It also defines common functions to add or remove entities. Additionally, it includes common attributes such as count, and capacity which will be relevant to all plots. It acts as a container which ensures organised placement and updating of entities over time.

FarmPlot (Inherits from Plot):

The FarmPlot class specifically manages plant entities in a farm plot. It is a container for a set of plant entities which records the specific information for each plant, the number of plants, and the capacity of the plot. When a plant becomes harvestable, it can be harvested with the according harvest method and adds the resulting item to the player's inventory.

Penn (Inherits from Plot):

Analogous to the FarmPlot class, the Penn class manages the animal entities in the game world. It is a container for a set of animal entities which holds the specific information for each animal, the number of animals, and the capacity of the Penn. Additionally, it monitors feeding and growth for animals and supports the slaughtering transition.

Shop

The shop contains items such as carrots, potatoes or animals which the player can then use to plant in their plots. This allows players to spend the money gained from selling items on something in the game. Other items may be added in the future such as additional plots.

Display:

This class is responsible for drawing all elements to the screen, it takes in an array of Plots and uses the coordinates and dimensions to draw them to the screen, it also takes the inventory class and displays the items on the bottom of the screen, as well as drawing the player character. This class will only handle components that are always shown on the screen, not components which pop up for various actions such as selling and buying. It will however allocate space on the screen for these actions.

GameManager:

The GameManager handles the main game loop including progressing time, updating entities, and managing interactions between the player, plots, and inventory. It ensures the transition of game-world entities, into inventory items when appropriate actions are taken by the player.

Data and Function Members

Item

Attributes:

int sellCost

int buyCost

bool plantable
string name

Functions

int getSellCost()
int getBuyCost()

string getName()

Plot

Attributes:

int currentCapacity

int maximumCapacity

tuple<int y, int x> topLeftCoord // Note, y is before x to be consistent with how ncurses functions accept coordinates

tuple<int height, int width> dimensions

Functions:

Int getCurrentCapacity()

Int getMaxCapacity()

bool setCurrentCapacity(int newCap) // returns false if above max

tuple<int y, int x> getTopLeftCoord()

tuple<int height, int width> getDimensions()

FarmPlot (Inherits from Plot):

Attributes:

Std::vector<Plant*> plants

Functions:

bool addPlant(Plant* plant) // returns false if full, true if succeeds , this also assigns some coordinates to the plant

Bool removePlant(tuple<int, int> position) // removes plant from the plot

Penn (Inherits from Plot):

Attributes:

Std::vector<Animal*> animals

Functions:

bool addAnimal(Animal* animal) // returns false if full, true if succeeds.

Bool removeAnimal(tuple<int, int> position) // If there is a grown animal, removes it from the plot

Entity:

Attributes:

Int growthStage
Int growthRequired

int caredForDays
int careRequired

Functions:

virtual void advanceDay()

bool isMature() const

Plant (Inherits from Entity):

Attributes:

Tuple<int, int> position

Functions:

Virtual void grow() = 0 (virtual function which has to be implemented in the individual plant classes as different plants have different grow requirements)

Void setPosition(tuple<int, int> pos)

Tuple<int,int> getPosition() const

Virtual Item harvest() = 0 // harvests plant and adds to user's inventory, removes itself from FarmPlot plants array. As the type of item added to the users inventory, the quantity, and harvest conditions vary, this is a virtual function.

CarrotPlant (Inherits from Plant):

Functions:

Void grow()

Item[2] harvest()

PotatoPlant (Inherits from Plant):

Functions:

Inherited methods from Plant

Void grow()

Item[4] harvest()

Animal (Inherits from Entity):

Attributes:

Tuple<int, int> position

Functions:

Virtual void raise() = 0 (virtual function which has to be implemented in the individual animals classes as different animals have different evolution requirements)

Void setPosition(tuple<int, int> pos)

Tuple<int,int> getPosition() const

Virtual Item slaughter() = 0 // harvests animal and adds to user's inventory, removes itself from Penn animals array. As the type of item added to the users inventory, the quantity, and slaughter conditions vary, this is a virtual function.

Cow (Inherits from Animal):

Functions:

Void raise();

Item[3] slaughter();

Player:

Attributes:

Tuple<int, int> position

Int direction // 0 for north, 1 for east, 2 for south, 3 for west

Inventory playersInventory

Int money

Functions:

Void move(tuple<int, int> newPosition)

Void water(*Plant)

Void feed(*Animal)

Item gather(*Entity)

Void plant(*Item, *FarmPlot)

Inventory* getInventory()

Inventory:

Attributes:

Std::map<Item, int> inventoryMap

Functions:

Void addItem(Item item, int Quantity)

Int searchItem(string name) // returns index of item or –1 if not found

Void removeItem(int index, int Quantity)

Int howMany(int index)

Void printInventory()

Display

Functions:

Void drawGameWindow()

Void drawInventoryWindow()

Void drawDynamicWindow()

Void drawCommands() // window displaying hotkeys required to do actions such as go to the next day

Void drawPlots()

Void drawPlayer(*Player)

Void drawInventory(*Player)

GameManager

Attributes:

Int currentDays;

FarmPlot** farmPlots;

Functions:

Void nextDay()

Void growIfTended()

Void handleUserInput()

Shop

Functions:

Bool buyCarrot(int qty, *Player player) // these methods will return false if the player does not have sufficient money to purchase them

Bool buyPotato(int qty, *Player, player)

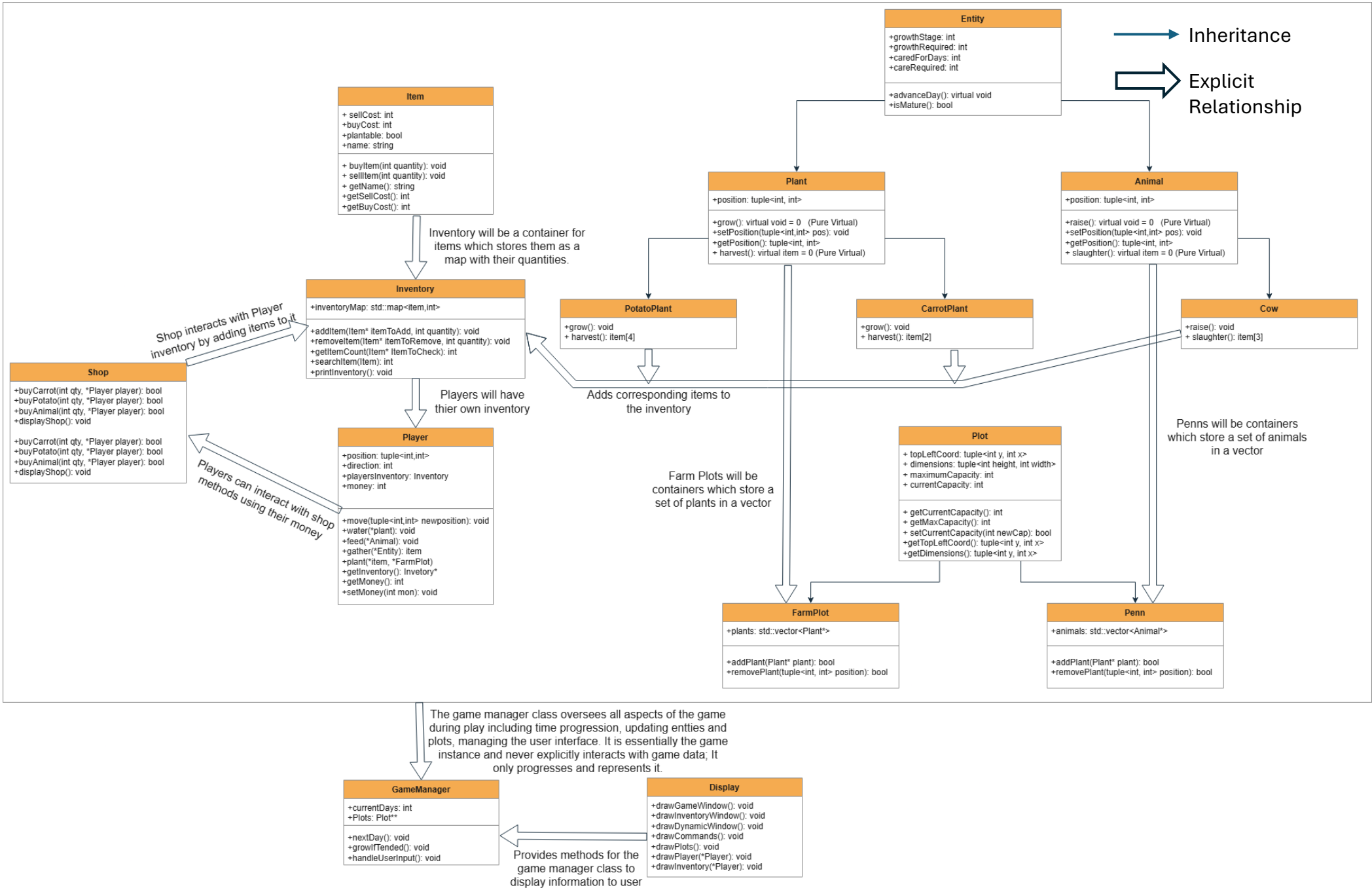
Bool buyAnimal(int qty, *Player, player)

Void displayShop() // Displays shop in the “dynamic window” space

Relationships between Classes

- Player contains an Inventory and interacts with Shop to buy items.
- Inventory holds a map of Items and provides methods to add, remove, and check items.
- Shop modifies the Player’s Inventory when purchases are made.
- Item is the base class used for all in-game items.
- Plant and Animal inherit from Entity.
- PotatoPlant and CarrotPlant inherit from Plant and override grow and harvest.
- Cow inherits from Animal and overrides raise and slaughter.
- FarmPlot contains a vector of Plants.
- Penn contains a vector of Animals.
- Plot is a base class for FarmPlot and Penn, providing shared functionality for capacity and dimensions.
- GameManager maintains the current day and a list of Plots.
- GameManager calls Display methods to render game information.
- Display provides rendering functions for Player and window output.

The UML class diagram below illustrates the above relationships between classes.


Entity
+growthStage: int
+growthRequired: int
+caredForDays: int
+careRequired: int
+advanceDay(): virtual void
+isMature(): bool
Inheritance
Explicit Relationship
Item
+ sellCost: int
+buyCost: int
+plantable: bool
+name: string
+ buyItem(int quantity): void
+ sellItem(int quantity): void
+ getName(): string
+getSellCost(): int
+getBuyCost(): int
Inventory will be a container for items which stores them as a map with their quantities.
Plant
+position: tuple<int, int>
+grow(): virtual void = 0 (Pure Virtual)
+setPosition(tuple<int,int> pos): void
+getPosition(): tuple<int, int>
+ harvest(): virtual item = 0 (Pure Virtual)
Animal
+position: tuple<int, int>
+raise(): virtual void = 0 (Pure Virtual)
+setPosition(tuple<int,int> pos): void
+getPosition(): tuple<int, int>
+ slaughter(): virtual item = 0 (Pure Virtual)
Inventory
+inventoryMap: std::map<item,int>
+addItem(Item* itemToAdd, int quantity): void
+removeItem(Item* itemToRemove, int quantity): void
+getItemCount(Item* ItemToCheck): int
+searchItem(Item): int
+printInventory(): void
PotatoPlant
+grow(): void
+ harvest(): item[4]
CarrotPlant
+grow(): void
+ harvest(): item[2]
Cow
+raise(): void
+ slaughter(): item[3]
Shop interacts with Player inventory by adding items to it
Shop
+buyCarrot(int qty, *Player player): bool
+buyPotato(int qty, *Player player): bool
+buyAnimal(int qty, *Player player): bool
+displayShop(): void
+buyCarrot(int qty, *Player player): bool
+buyPotato(int qty, *Player player): bool
+buyAnimal(int qty, *Player player): bool
+displayShop(): void
Players will have thier own inventory
Adds corresponding items to the inventory
Player
+position: tuple<int,int>
+direction: int
+playersInventory: Inventory
+money: int
+move(tuple<int,int> newposition): void
+water(*plant): void
+feed(*Animal): void
+gather(*Entity): item
+plant(*item, *FarmPlot)
+getInventory(): Invetory*
+getMoney(): int
+setMoney(int mon): void
Players can interact with shop methods using their money
Farm Plots will be containers which store a set of plants in a vector
Plot
+ topLeftCoord: tuple<int y, int x>
+ dimensions: tuple<int height, int width>
+ maximumCapacity: int
+ currentCapacity: int
+ getCurrentCapacity(): int
+ getMaxCapacity(): int
+ setCurrentCapacity(int newCap): bool
+getTopLeftCoord(): tuple<int y, int x>
+getDimensions(): tuple<int y, int x>
Penns will be containers which store a set of animals in a vector
FarmPlot
+plants: std::vector<Plant*>
+addPlant(Plant* plant): bool
+removePlant(tuple<int, int> position): bool
Penn
+animals: std::vector<Animal*>
+addPlant(Plant* plant): bool
+removePlant(tuple<int, int> position): bool
The game manager class oversees all aspects of the game during play including time progression, updating entties and plots, managing the user interface. It is essentially the game instance and never explicitly interacts with game data; It only progresses and represents it.
GameManager
+currentDays: int
+Plots: Plot**
+nextDay(): void
+growIfTended(): void
+handleUserInput(): void
Display
+drawGameWindow(): void
+drawInventoryWindow(): void
+drawDynamicWindow(): void
+drawCommands(): void
+drawPlots(): void
+drawPlayer(*Player): void
+drawInventory(*Player): void
Provides methods for the game manager class to display information to user

Project Task List and Timeline

1. Design class structure and relationships (1-2 days) (All group members)
2. Implement Item, Inventory, and Player classes (1 day) (Archie, Praj)
3. Implement Entity, Plant, Animal, and their derived classes (2 days) (Arkaintz, Archie)
4. Implement Plot, FarmPlot, and Penn classes (1 day) (Praj)
5. Implement the Shop class and purchase logic (1 day) (Archie)
6. Implement the GameManager, and Display classes (2 days) (Praj, Arkaintz)
7. Implement main game loop and player commands (2 days) (All group members)
8. Write unit tests and test game interactions (1 day) (Archie)
9. Debug and add exception handling (2 days) (All group members)
10. Write user documentation and prepare for submission (All group members)

User Interaction Description

The game interface utilizes the ncurses library to create a fully keyboard driven, interactive text-based environment in the terminal. The user interface provides real-time feedback, and dynamically adapts based on the player's actions, location, and other context-based information to help players navigate the game without any confusion.

Players interact entirely via the keyboard. Common movement keys such as the "WASD" keys are used for movement, while specific keys (e.g, B to buy, P to plant) trigger contextual actions. To ensure clarity and ease of usability, a dynamic keybind display will be implemented at the top of the screen. This panel will update automatically to reflect the actions available to the player based on their current scenario.

The terminal user interface itself is divided into 4 distinct sections:

- **Top**: A dynamic Keybind display area that updates based on the player's available actions.
- **Center**: A visual representation of the game world which displays the player, their virtual farm, including farm plots, Penns, planted plants, paths, and structures like the shop. The player's character can be moved around in this environment using the assigned directional keys.
- **Right side**: A portion of the screen on the right side will be reserved for location specific actions. For example:
    - Near the Shop: Displays the items available for purchase.
    - Near a farm plot: Displays information about the plot and gives the ability to plant crops depending on where the player's standing.
- **Bottom:** The bottom of the terminal is reserved for the player's inventory and current funds, which are always visible during gameplay.

Additionally, the game will provide immediate feedback for "illegal" actions, such as trying to buy more items than they can afford, through clear on-screen error messages. An error message will also display if the terminal window is too small to properly display the game, prompting the user to resize their window which ensures that the interface is rendered correctly. The overall user interaction design prioritizes clarity, engagement and usability to deliver a smooth user experience despite the limitations of a terminal-based interface.

Unit Testing and Debugging Plan

In ensuring a reliable and robust approach to compile-time unit testing and debugging, Google's framework Google Test will be integrated and applied as a build step. For runtime debugging, C++'s assertions will be used to validate function parameters and application state. When a bug is found in the codebase through playtesting or code review, it will be raised as an open issue in the project's GitHub repository. As the project progresses, issues on GitHub will be rectified in order of descending importance and covered in future unit testing (if eligible) to ensure the issue doesn't resurface. For nonfatal runtime errors, C++'s iostream library will be used to make the error readily apparent via logging without yielding the application unnecessarily.

The project will be generated using the GNU Make build system along with a custom Makefile which includes compilation, linking, and Google Test related commands. Additionally, the Makefile will include multiple build targets – i.e. Debug x64, Release x64, Debug x86, etc, and comprehensive documentation related to the compilation and dependency linking process. In addition to the Makefile, the project will have user documentation stored in a README markdown file which make the compilation process and execution process as straightforward as possible.

Additionally, to coincide with the project's object-oriented structure, the unit tests required are categorized on a class-by-class basis. Possible necessary tests have been planned below.

FarmPlot

Test 1. Run FarmPlot::addPlant FarmPlot::getMaxCapacity() + 1 times with a valid *plant* parameter value, expecting *true* for every call except for the final call, where *false* is expected instead.

Test 2. Run FarmPlot::getCurrentCapacity and FarmPlot::addPlant Plot::getMaxCapacity() + 1 times expecting the i_{th} call to return i, except for the final call which should return $i - 1$.

Penn

Test 1. Run Penn::addAnimal Penn::getMaxCapacity() + 1 times with a valid animal pointer, then call expecting *true* for every call except for the final call, where *false* is expected instead.

Test 2. Run Plot::getCurrentCapacity and Penn::addAnimal Plot::getMaxCapacity() + 1 times expecting the i_{th} call to return i, except for the final call which should return $i - 1$.

Item

Test 1. Run Item::getName expecting Item::name

Test 2. Run Item::getSellCost expecting Item::sellCost

Test 3. Run Item::getBuyCost expecting Item::buyCost

Entity

Test 1. Run Entity::advanceDay Entity::growthRequired + 1 times expecting Entity::growthStage to increase each time, except for the last call which should not have an effect.

Test 2. Expect Entity::isMature() to return Entity::growthStage >= Entity::growthRequired

Plant

Test 1. Expect Plant::Plant() (or equivalent constructors) to only be callable from a derived type.

Shop

Test 1. Expect Shop::buyCarrot(1, playerPtr) to return playerPtr->money >= carrotCost where carrotCost is the cost of a single carrot item and playerPtr is a valid pointer to a player object.

Test 2. Expect Shop::buyPotato(1, playerPtr) to return playerPtr->money >= potatoCost where potatoCost is the cost of a single potato item and playerPtr is a valid pointer to a player object.

Test 3. Expect Shop::buyAnimal(1, playerPtr) to return playerPtr->money >= animalCost where animalCost is the cost of a single animal and playerPtr is a valid pointer to a player object.

Inventory

Test 1. Call Inventory::searchItem(item->getName()), expecting -1, then Inventory::addItem(item, 1) and Inventory::searchItem(item->getName()), expecting the latter to return 0. Then, call Inventory::removeItem(0, 1) and Inventory::searchItem(item->getName()), expecting the latter to return -1 once again.