
FARM SIMULATOR

PROJECT REPORT
OBJECT ORIENTED PROGRAMMING AND MODELING

CREATED BY
STEFANO ZIZZI, LUIS FRASHERI

*Università degli Studi di Urbino
Informatica Applicata*



BADGE NUMBER: 312793, 312972
JUNE 2023

Contents

1	Analysis	5
1.1	Objectives	5
1.2	Requirements	6
1.2.1	Functional Requirements	6
1.2.2	Non-functional Requirements	6
1.3	Domain Model	6
2	Design	7
2.1	Architecture	7
2.1.1	Model	7
2.1.2	View	7
2.1.3	Controller	8
2.2	Detailed Design	8
2.2.1	Stefano Zizzi	8
2.2.1.1	Actions	8
2.2.1.2	GameBackup	10
2.2.1.3	PlantLand	11
2.2.1.4	PlantChunk	12
2.2.1.5	Plants	12
2.2.2	Luis Frasheri	13
3	Development	15
3.1	Automated Testing	15
3.1.1	Luis Frasheri	15
3.2	Working Method	15
3.2.1	Workflow	15
3.2.2	Stefano Zizzi	16
3.3	Development Notes	16
3.3.1	Luis Frasheri	16
3.3.2	Stefano Zizzi	16

Chapter 1

Analysis

1.1 Objectives

The Team aims to develop an application for playing a farm simulator, hence its name "**FarmSimulator**". Through the possibility of exchanging between two roles, namely:

- **Farmer,**
- **Landlord,**

the game provides the opportunity to simulate the experience of a farm in different ways. Using the Farmer, it is possible to manage the land, possibly populated by animals or plants, and take care of them so that they produce the products. Through the Landlord, it will instead be possible to sell and buy products, as well as land.

The lands that host animals or plants are divided into smaller pieces of land called "**Chunk**". In the case of plants, each Chunk has its level of fertilization and humidity that will have an impact on the growth rate of the plant, if the plant is not watered for seven days straight, it will dry up and die. The present plants are:

- **Carrot;**
- **Onion;**
- **Potato;**
- **Wheat.**

The growth of animals depends on their level of thirst and hunger. If the animal receives the right care, it will produce more products. Otherwise, it will die. The present animals are:

- **Chicken;**
- **Cow;**
- **Pig;**
- **Goat.**

The farmer does not have access to the market since his role is to manage the lands, just as the owner cannot access the lands. The use of objects for treating animals, plants, or land may be necessary, but it may also be used only if you want to obtain bonuses. The objects available to the Farmer are:

- **Hoe;**
- **Scissors;**
- **Scycle;**
- **Fertilizer;**
- **Watering Can.**

Thanks to the possibility of saving, loading, and deleting progress, there is no actual limit to the game time.

1.2 Requirements

1.2.1 Functional Requirements

- The user can assume the role of owner and farmer;
- The user, through the role of owner, can buy and sell portions of land, as well as the necessary resources;
- The user, through the role of farmer, must work on the portion of land to allow the growth of plants or the breeding of animals, and will have access to a personal inventory;
- The application will allow each portion of land to host animals or plants;
- The application will have a repository for the storage of tools, resources, and animals;
- The application will have a shop from which to purchase seeds, resources, animals, or land;
- The application will manage the variation of seasons that influence the weather;

1.2.2 Non-functional Requirements

- The serialization of games for their respective file saving and loading must be efficiently executed, and avoid any file corruption issues;
- The application is structured in such a way that adding any locations, roles, actions, animals or plants is as simple as possible.

1.3 Domain Model

Farm Simulator will contain the **Game**, which holds all the information regarding the game:

- The current player, of type **Person**, who will alternate between Farmer and Landlord;
- The **Calendar** that manages the passing of days and the variation of seasons and weather;
- The lands of type **LandAbstract**, which are divided into **AnimalLand** and **PlantLand**, which in turn are divided into **Chunk**.
- The Barn that contains an additional **Inventory**, as well as the **Market** where the Landlord can buy and sell objects;

The objects are called **Items**, each of which has its own price, type, and number. Items can be overlapped up to a maximum defined within the Item. The type of object is divided between Plant, Animal, **Product**, and Tool, the latter unlike the others, has a durability that determines its destruction once finished.

One of the main difficulties was managing the actions available to the player based on the **Place**, role, or equipped Tool. All this was executed through the use of an **ActionManager** that manages all possible **Actions** in the game.

Chapter 2

Design

2.1 Architecture

For the development of FarmSimulator's architecture we decided to use the design pattern MVC(Model-View-Controller). This architecture separates the game logic from the graphical user interface (GUI), which improves modularity and simplifies development.

2.1.1 Model

The Model represents the game itself, so it edits game data, manages interactions and updates the state of the game. Its aim is to give the Controller a way to interact with the game. This implementation the implementation of the game backup really trivial. We designed the Model with a data-driven approach, which separates the game data from the game logic. This allows for easy serialization of game data, making it easy to save and load game progress.



Figure 2.1: UML Diagram of the Model

2.1.2 View

The View is the application screen that manages the graphical part, User Experience, and interaction with the user. It is the task of the various Views to record and inform their respective Controller of user interactions with the application, waiting for its response on data changes. The View was made by using Java Swing. By isolating the Controller from the framework used, we have made sure that changing the graphics library only requires rewriting the View part, leaving the Controller and Model part unchanged. The View was divided in different classes, each representing a different zone of the game.

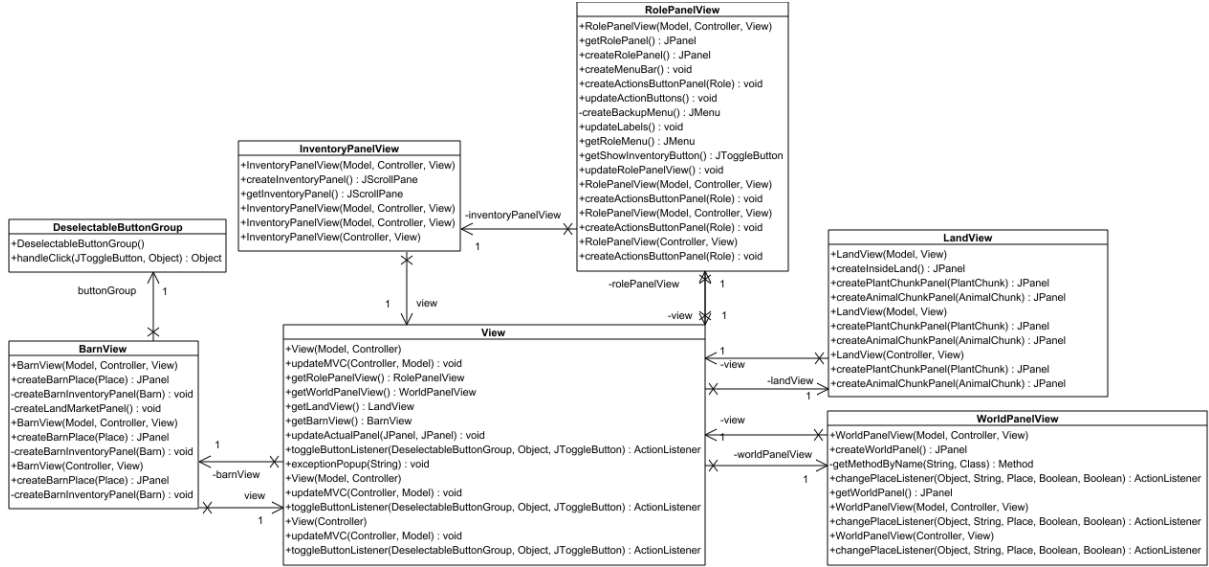


Figure 2.2: UML Diagram of the View

2.1.3 Controller

The Controller is the component responsible for managing user interactions in the View in a consequential manner, thus communicating the change to the Model. Once the Model has completed processing the change request, the Controller will notify its View, so that the latter can update consistently according to the rules specified by the Model.



Figure 2.3: UML Diagram of the Controller

2.2 Detailed Design

2.2.1 Stefano Zizzi

One problem we had to tackle from the beginning was managing the actions available to the player, as they not only varied by role but also by the location they were in. For this reason, a significant part of the project's development was dedicated to creating an action system that was also easily expandable and adaptable in case new locations, roles, or specific actions were to be added.

2.2.1.1 Actions

After some considerations, it was clear that we needed an entity that managed the addition and removal of actions from the player and that every player and location had their set of actions to update.

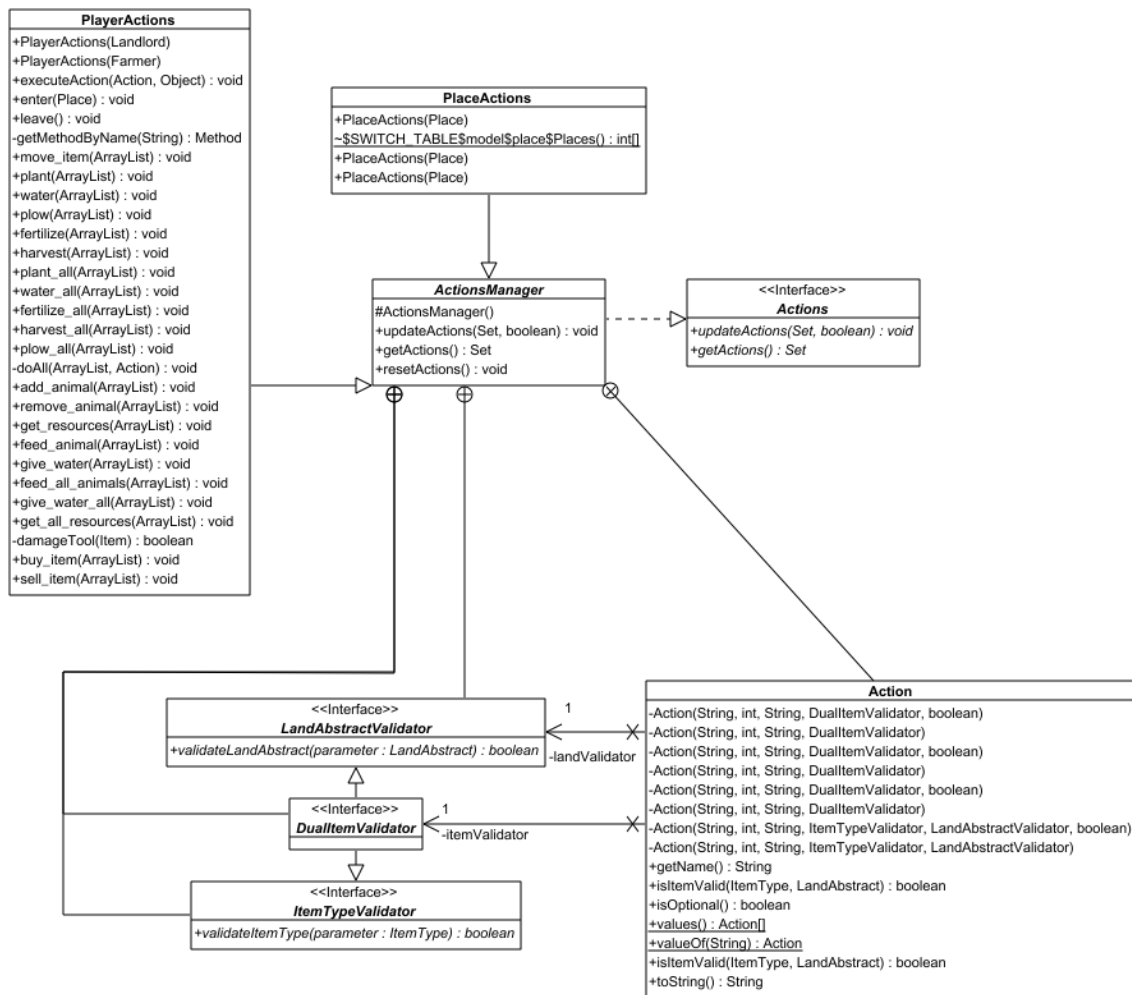


Figure 2.4: UML Diagram of the Actions

ActionsManager is an abstract class that implements the **Actions** interface and provides a series of useful methods and attributes for managing the actions of the game characters.

The attributes of this class include a set of available actions (`availableActions`) that is used to keep track of the possible actions that game characters can take. This set is initialized with a `HashSet` object inside the protected **ActionsManager()** constructor.

The **ActionsManager** class also includes an enumeration called **Action** that defines the possible actions that game characters can take. Each action has a name (`name`), an item type validator (`itemValidator`), and a land validator (`landValidator`). The item type validator is used to determine if the object used to perform the action is valid for the action itself. The land validator is used to determine if the location where the action is being performed is valid for the action itself. The **Action** enumeration also includes an optional flag (`optional`) that indicates if the action can be taken even without using an object.

The **ActionsManager** class also provides several useful methods for managing actions. For example, the `updateActions` method is used to update the set of available actions. The **ActionsManager** class also includes three interfaces (**ItemTypeValidator**, **LandAbstractValidator**, and **DualItemValidator**) that define the methods used to determine if an object or a location is valid for an action.

In summary, **ActionsManager** is an abstract class that provides a series of useful methods and attributes for managing the actions of game characters. Its subclasses can use these methods and attributes to implement specific actions for their game.

I decided to use a `HashSet` that is contained in both places and people.

Place Actions `PlaceActions` extends the abstract class `ActionsManager` and provides a series of useful methods and attributes for managing the actions that can be taken in a specific location (`Place`) within the game.

The attributes of this class include a reference to the `Place` object where the action is being performed (`place`). The `PlaceActions` constructor accepts a `Place` parameter and initializes the `place` attribute with the passed value.

The `PlaceActions` constructor also defines the set of available actions for the specific `Place` based on the type of `Place`. Depending on the type of `Place`, a new `HashSet` of `Action` is created, and the available actions are added to the `HashSet` based on the type of `Place`. For example, if the `Place` is a type of `Plant Land`, only the `FLOW_ALL` action is set, whereas if the `Place` is a type of `Barn`, only the `MOVE_ITEM` action is set, and so on. The `Action` enumeration is inherited from the `ActionsManager` class.

The `PlaceActions` class also inherits the methods of the `ActionsManager` class to manage the set of available actions.

Player Actions All actions available to the player are contained within the `PlayerActions` class. Having to find an easy way to call them, without having to call the specific method by hand, I thought of using the `Reflect` library to be able to call the methods through a string (ie their name), matching the names of the actions contained in the enumeration of `ActionManager` with method names.

Has two constructors, one for `Landlord` and one for `Farmer`, both take an instance of the `Person` object as an argument. The class also has two attributes: `"person"`, which represents the player, and `accessiblePlaces`, which contains the places accessible by the player.

The `executeAction` method executes an action specified as an argument by calling the corresponding method in this class. If the method exists and is available, it is called, otherwise an `"ActionNotAvailableException"` is thrown.

The `enter` method changes the player's actions when entering a location, checking that the location is one of the player's accessible locations. If the location is accessible, the player's current location is changed and the available actions are updated. If the player is a `"Landlord"` and is in a `"Barn"`, he cannot move objects.

The `leave` method changes the player's actions when leaving a location, removing the available actions for the current location.

There are also several methods for game specific action, such as `"move_item"`, `"plant"`, `"water"`, `"plow"`,... which perform the user specified action on a specific object. All of these methods check that the item is available for the specified action and that the player has the necessary tools.

In addition, there are `"doAll"` methods that repeat a specified action for all elements within a given location. Methods check if the action is available for each element and if the player has the tools needed to perform the action.

2.2.1.2 GameBackup

During the development of the project we decided that a useful feature would be to be able to save the game progress and reload it at any time.

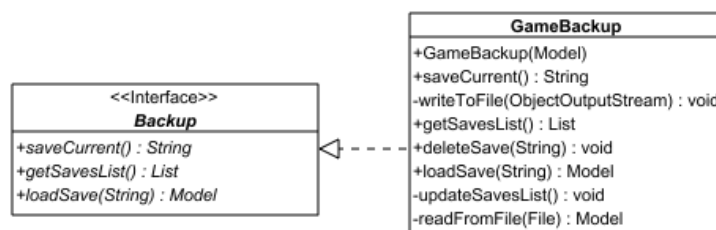


Figure 2.5: UML Diagram of the GameBackup

The `GameBackup` class represents an implementation of the `Backup` interface and provides methods for saving, loading, and deleting game saves.

The constructor accepts an instance of the `Model` class, which represents the state of the game to be saved.

The `saveCurrent()` method creates a save file with the format name `dd_MM_yyyy-HH_mm_ss` (date-time format) in the `saves` folder and saves the game state in this file. The method returns the name of the newly created save file.

The `getSavesList()` method returns the list of save files present in the `saves` folder.

The `deleteSave(String saveName)` method deletes the specified save file.

The `loadSave(String saveName)` method loads the game state from a specified save file. If the file does not exist or is corrupted, an `IOException` or `ClassNotFoundException` is thrown.

The `GameBackup` class uses the `writeToFile()` and `readFromFile()` methods to write and read data from the save file.

In general, the `GameBackup` class provides essential functionality for saving and restoring game state, allowing players to stop and resume the game at a later time. For this class to work it was necessary to use serialization, so each object of the model to be saved implements the `Serializable` interface.

2.2.1.3 PlantLand

This class represents a type of land that contains chunks of land with plants. It extends the abstract class `LandAbstract` and implements the `Iterable` interface to iterate over its plant chunks.

The class has an `ArrayList` of `PlantChunk` objects to store the chunks of land with plants. Its constructor initializes the type of the land as `Places.PLANT_LAND`, creates ten `PlantChunk` objects and adds them to the `ArrayList`.

The class has a method `getNumElements()` that returns the number of chunks of land that have plants. It iterates through the `ArrayList` and checks if each chunk has a plant. If a chunk has a plant, it increments the elements counter and returns its value.

The `update()` method updates all the chunks in the land by calling the `update()` method of each `PlantChunk` object. It also checks if there are harvestable plants in any of the chunks. If there are no harvestable plants or no elements in the land, it updates the actions of the land with the `HARVEST_ALL` action disabled. Otherwise, it updates the actions with the `HARVEST_ALL` action enabled.

The `getElements()` method returns the `ArrayList` of `PlantChunk` objects.

Finally, the class implements the `iterator()` method of the `Iterable` interface to allow iteration over its chunks of land with plants.

2.2.1.4 PlantChunk

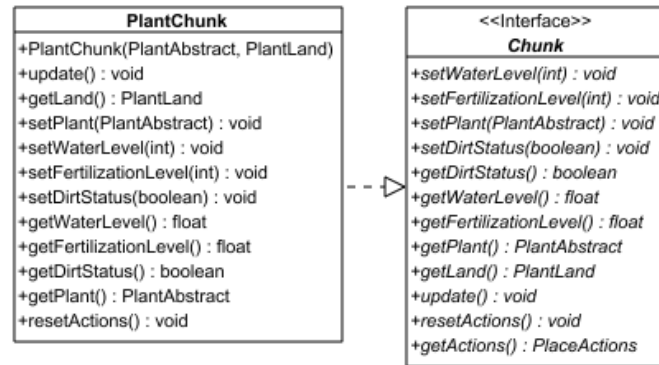


Figure 2.6: UML Diagram of the PlantChunk

The **PlantChunk** class represents a small part of land that contains one plant. It has several attributes, including a **PlantAbstract** object that represents the plant on the chunk, a reference to the **PlantLand** object that the chunk belongs to, and variables for tracking the water and fertilization levels of the chunk.

One of the main methods in the **PlantChunk** class is the `update()` method, which is called periodically to update the status of the chunk. This method grows the plant on the chunk, decreases the water and fertilization levels, and updates the available actions for the chunk based on the status of the plant.

Other important methods in the class include `setPlant()`, which sets the plant on the chunk, `increaseWaterLevel()` and `increaseFertilizationLevel()`, which increase the water and fertilization levels of the chunk respectively, and `resetActions()`, which resets the available actions for the chunk based on its plowed status.

Overall, the **PlantChunk** class plays an important role in the game by representing a small part of land that can be used to grow plants. It has several methods that are used to update the status of the chunk and the plant on it, as well as to manage the available actions for the chunk.

2.2.1.5 Plants

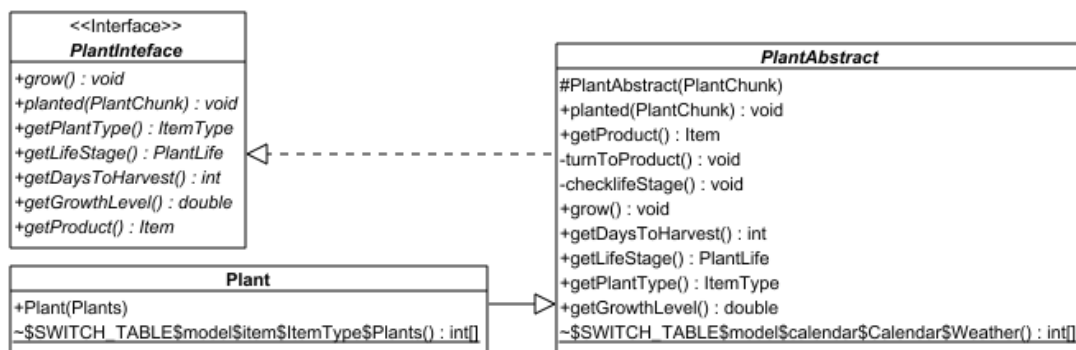


Figure 2.7: UML Diagram of the Plants

The **PlantAbstract** class is an abstract class that serves as the parent class for all types of plants in the game. It implements the **PlantInterface** interface, which defines the methods that must be implemented by all plant classes.

The class contains several attributes such as `daysToHarvest`, `lifeStage`, `growthRate`, `maxGrowthLevel`, `daysWithoutWater`, `chunk`, `calendar`, and `random`. It also defines an enum called **PlantLife** which represents the different stages in a plant's life cycle.

The class has several methods to perform actions on the plant, such as `planted()` which sets the chunk where the plant was planted, `getProduct()` which returns all the products the plant has produced, `turnToProduct()` which turns the plant to a product when harvested, `checkLifeStage()` which

changes the life stage of the plant.

The `grow()` method in 'PlantAbstract' is responsible for updating the growth of the plant, which depends on several factors, such as the water and fertilization levels, the weather, and the plant's growth rate. The method first calculates a growth factor based on these conditions, and then uses that factor to increment the growth level of the plant.

If the plant's growth level reaches certain thresholds, its life stage is updated to reflect its current state, such as going from a seed to a sprout or from an adult plant to a harvestable plant.

The method also checks if the plant has not received water for a certain number of days, and if so, increments a counter. If the counter reaches a certain value, the plant is considered dead and its life stage is updated accordingly.

Overall, the `grow()` method simulates the growth of the plant over time and takes into account various conditions that affect its growth, ensuring a realistic and engaging gameplay experience.

Other methods include `getDaysToHarvest()` which returns the plant's remaining days until harvest, `getLifeStage()` which returns the plant's life stage, `getPlantType()` which returns the species of the plant, and `getGrowthLevel()` which returns the plant life.

Overall, the `PlantAbstract` class provides a foundation for all plant classes and defines common methods and attributes that can be used by them.

2.2.2 Luis Frasheri

Item

Inventory

Barn

Chapter 3

Development

3.1 Automated Testing

Automated tests have been used to make the work as efficient as possible, especially when all classes were interconnected. JUnit 4.13.2 was used to implement the automated tests. Most of the tests performed are generic, meaning that each of these tests goes through the entire process that will lead to the result, such as the production of Products from animals. Of course, there are also some specific tests to test those parts of the code that are somewhat sensitive and have led to various errors over time.

3.1.1 Luis Frasheri

Considering that the tests were performed with a slight delay, we decided to carry out some general tests, such as 'plantedToHarvestPlant' and 'addedAnimalToChunkAndGetProducts', in order to test small implementations through larger implementations, thus testing all functions.

Afterwards, critical functionalities were tested, such as product stacking, purchase and sale, which involved continuous exchanges between various inventories.

Another critical point of the project was the saving feature, through which it was possible to interact with the software and corrupt the data.

- plantedToHarvestPlant
- addedAnimalToChunkAndGetProducts
- productsShouldBeStacked
- boughtItemShouldBeAddedToTheBarn
- balanceShouldBeGreaterAfterSell
- landShouldBeAddedAfterBoughtIt

3.2 Working Method

The closeness of the group members allowed for the development of the code mainly through live meetings held almost daily. In the initial phase of the project, we focused on creating a UCD that would give us an overview of the program. This phase helped us divide the work and ensure independent work for each member (despite being close). Subsequently, we created the most comprehensive UML possible, which, despite being revised along the way, helped us greatly during the implementation phase.

3.2.1 Workflow

As the team is composed of only two individuals, with the opportunity to work closely together and the project being of small to medium size, we have decided to use the Crystal Clear development methodology. Crystal Clear was a suitable choice for our team as it promoted the creation of a cohesive and highly collaborative team. Thanks to the emphasis placed on effective communication and teamwork, we were able to stay aligned on our objectives and quickly solve any issues that arose.

3.2.2 Stefano Zizzi

Implementation of the entire action management system. Features for saving to file and loading progress, avoiding file corruption issues.

Creation of the plant system as well as the chunk framework later used for the lands.

General management of the MVC structure.

3.3 Development Notes

3.3.1 Luis Frasheri

3.3.2 Stefano Zizzi

- **Functional Programming** during the development of this project, I was able to leverage it for what is related to the model interaction with the player.
- **Lambda Expressions** With functional programming, the use of lambda expressions has become necessary, if not mandatory, for stylistic purposes. They allowed me to have clean, readable, and clear code.
- **Streams**, widely used with PlayerActions, make use of functional programming and lambda expressions. I found them to be one of the most interesting features that Java offers.
- **Java Util Reflect** was used in both the view and Player Actions, it made view less repetitive and Player Actions very modular and expandible.
- **Maven** was a comprehensive tool that made building and testing really easy.

These notes have been prepared with L^AT_EX. The PDF is equipped with hyperlinking, so it is possible to click on section names in the index to move between pages. Similarly, to return to the index, simply click on the page number.