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 . . . 11
    - 2.2.1.3 PlantLand . . . 12
    - 2.2.1.4 PlantChunk . . . 12
    - 2.2.1.5 Plants . . . 13
  - 2.2.2 Luis Frasheri . . . 14
    - 2.2.2.1 Person . . . 14
    - 2.2.2.2 Calendar . . . 14
    - 2.2.2.3 Item . . . 14
    - 2.2.2.4 Inventory . . . 17
    - 2.2.2.5 Barn . . . 18

3 Development **21**
- 3.1 Automated Testing . . . 21
  - 3.1.1 Luis Frasheri . . . 21
- 3.2 Working Method . . . 21
  - 3.2.1 Workflow . . . 21
  - 3.2.2 Luis Frasheri . . . 22
  - 3.2.3 Stefano Zizzi . . . 22
- 3.3 Development Notes . . . 22
  - 3.3.1 Luis Frasheri . . . 22
  - 3.3.2 Stefano Zizzi . . . 22

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**;
- **Sickle**;
- **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 as well as a different material which defines its durability.
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` rapresents the game itself, so it edits game data, manages interactions and updates the state of the game. It's aim is to give the Controller a way to interact with the game. This made 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.

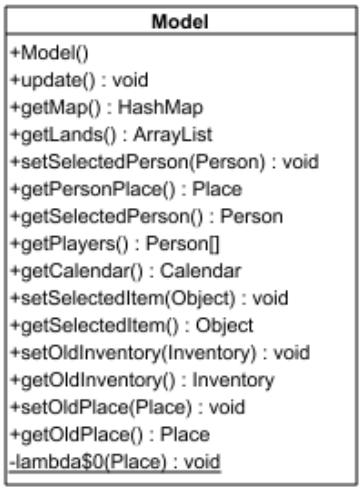

Figure 2.1: UML Diagram of the Model

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 devided in different classes, each rapresenting a different zone of the game.

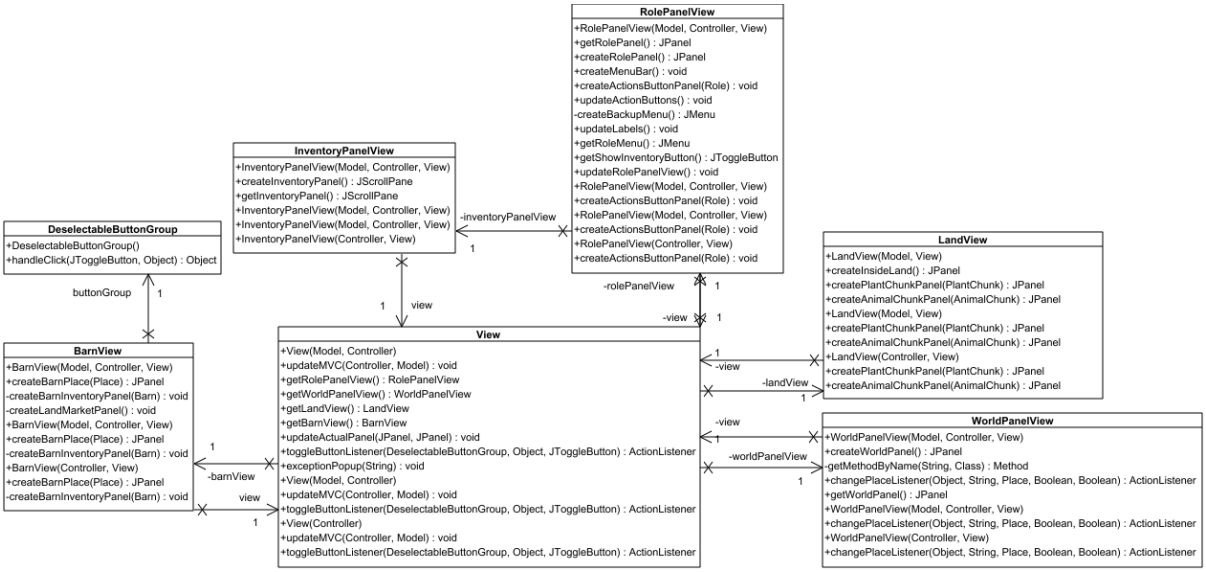

Figure 2.2: UML Diagram of the View

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.

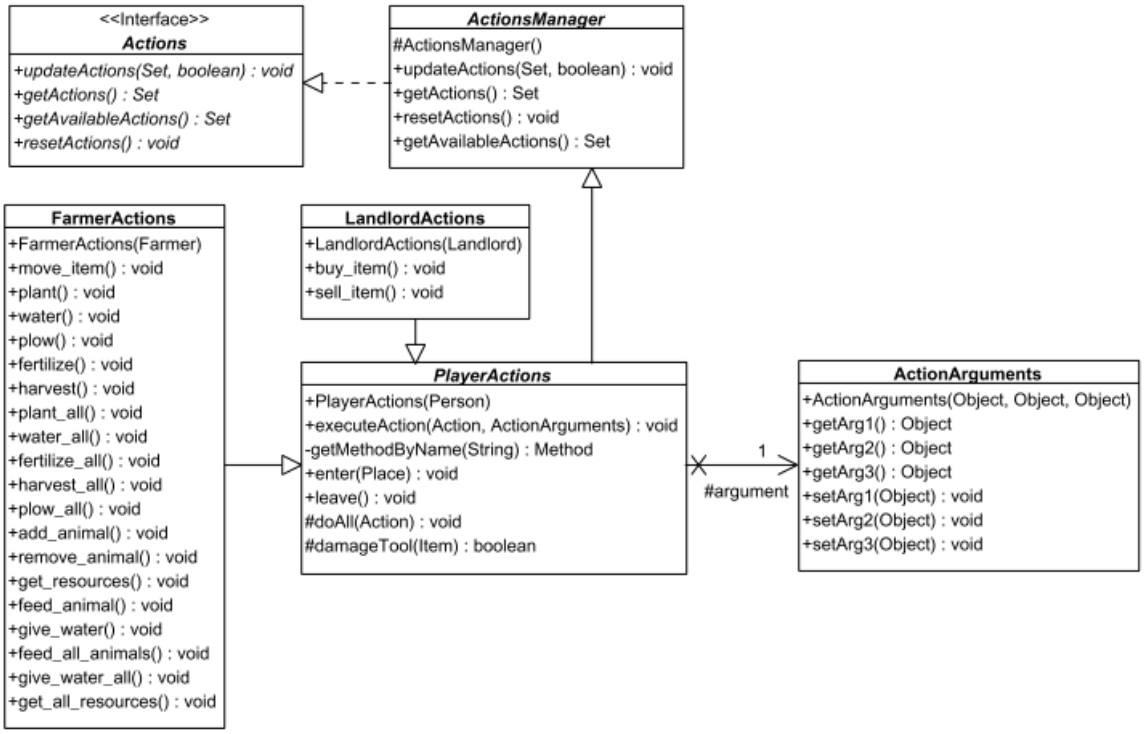

Figure 2.4: UML Diagram of the Actions

`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, along with the PlayerActions class, can be used to implement the Mediator design pattern. The Mediator pattern is used to manage communication between objects by encapsulating it in a mediator object. In this case, the mediator object would be the ActionsManager, which would handle the communication between the different game characters and their actions. This allows for loose coupling between the game characters and their actions, making it easier to modify and extend the game logic.

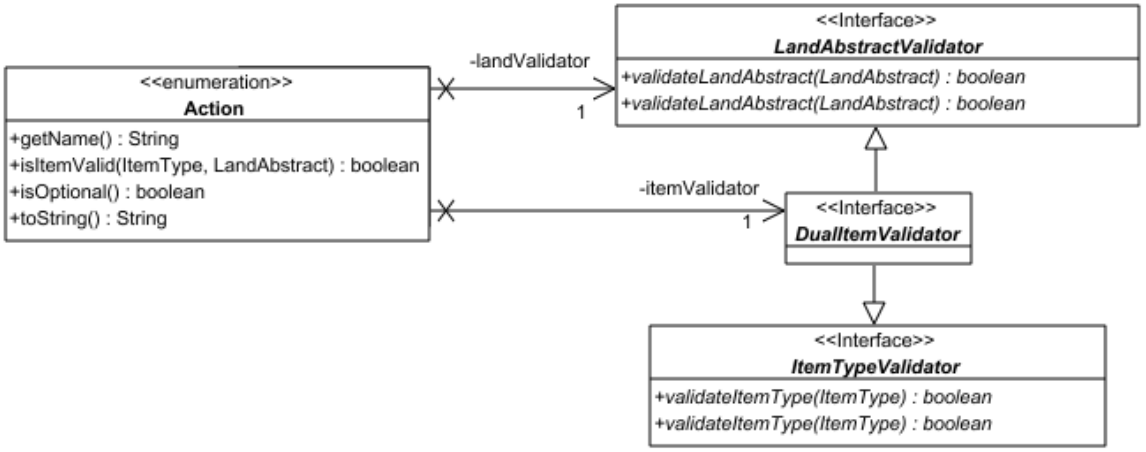

Figure 2.5: UML Diagram of the Actions Enumeration

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`), that are both Functional Interfaces. 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 `DualItem Validator`) 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 `PLOW_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.

The `PlayerActions` class is an abstract class that contains all the actions that a player can perform. The class extends the `ActionsManager` class and is generic over the type `T`, which represents a `Person` object. The class contains several attributes, including the person, which is the person performing the action, an argument of generic type `ActionArguments`, and a hashmap that maps a Role enum value to a corresponding Person subclass. The constructor of the class initializes the person and the `roleClass` attributes.

The class provides a method `executeAction` that takes an action and an argument and executes the corresponding method if it exists and is available. The class also provides a private method `getMethodByName` that returns a method by its name, either from the current class or from the role class. Additionally, the class provides methods `enter` and `leave` to change the actions when the actor enters or leaves a place.

The class also provides a method `doAll` that repeats an action on all the chunks of the land. The class also provides a method `damageTool` that damages a tool and removes it if it's worn out. The class handles several exceptions such as `ActionNotAvailableException, PlaceNotAvailableException, NoItemFoundException`, and `NotEnoughItemsException`.

In summary, the `PlayerActions` class represents a set of actions that a player can perform, provides

methods to execute these actions, and handles several exceptions that can occur during the execution of these actions. The class is abstract and generic over a Person object to provide a common interface for all the Person subclasses that can perform these actions.

LandlordActions The `LandlordActions` class represents the set of actions that a landlord can perform. The class extends the `PlayerActions` class and is specific to the Landlord subclass. The constructor of the class initializes the person attribute with the landlord parameter.

The class provides two methods, `buy_item` and `sell_item`, that allow the landlord to buy or sell an item or a land from or to the market. The `buy_item` method first checks if the item is a land or not. If it is a land, the method checks if the landlord can afford it and adds it to the lands list. If it is not a land, the method adds the item to the barn and removes the item's price from the landlord's balance. If the item is not available in the market, the method throws a `CannotBuyItemException`.

The `sell_item` method also checks if the item is a land or not. If it is a land, the method removes the land from the lands list and adds the land's sell price to the landlord's balance. If it is not a land, the method removes the item from the barn and adds the item's price to the landlord's balance. If the item is not available in the barn, the method throws a `CannotSellItemException`.

In summary, the `LandlordActions` class represents the set of actions that a landlord can perform, provides methods to buy or sell an item or a land from or to the market, and handles several exceptions that can occur during the execution of these actions. The class is specific to the Landlord subclass and extends the `PlayerActions` class to provide a common interface for all the Person subclasses that can perform these actions.

FarmerActions `FarmerActions` extends the `PlayerActions` class with a type parameter of "Farmer". The class contains methods for actions that can be performed by a farmer, such as moving an item to the barn, planting a seed, watering a plant, plowing a chunk, fertilizing a plant, harvesting a plant, adding an animal to the animal chunk, removing an animal from the animal chunk and adding it to the inventory, getting resources from the animal in the animal chunk, feeding an animal, and giving water to an animal.

The methods perform different actions, such as moving an item between the barn and the farmer's inventory, planting a seed in a chunk, watering a plant using a watering can, plowing a chunk using a hoe, fertilizing a plant using a fertilizer, harvesting a plant using a sickle or bare hands, adding an animal from the inventory to the animal chunk, removing an animal from the animal chunk and adding it to the inventory, getting resources from the animal in the animal chunk, feeding an animal, and giving water to an animal.

Overall, the `FarmerActions` class provides a set of actions that can be performed by a farmer in the game or simulation it is designed for.

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.

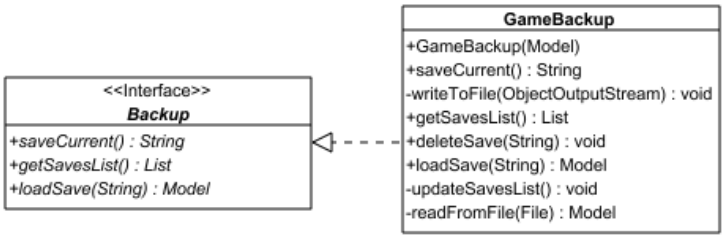

Figure 2.6: UML Diagram of the GameBackup

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 that had 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

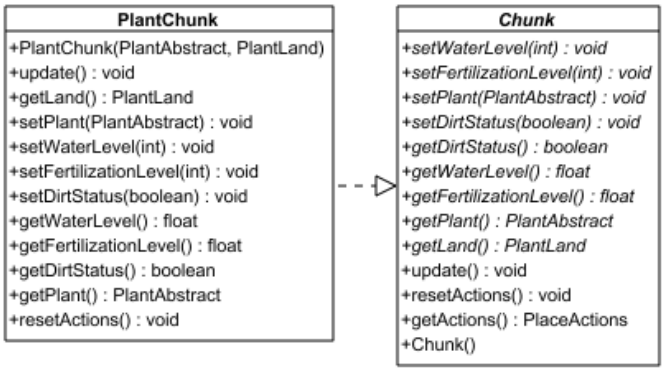

Figure 2.7: UML Diagram of the PlantChunk

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

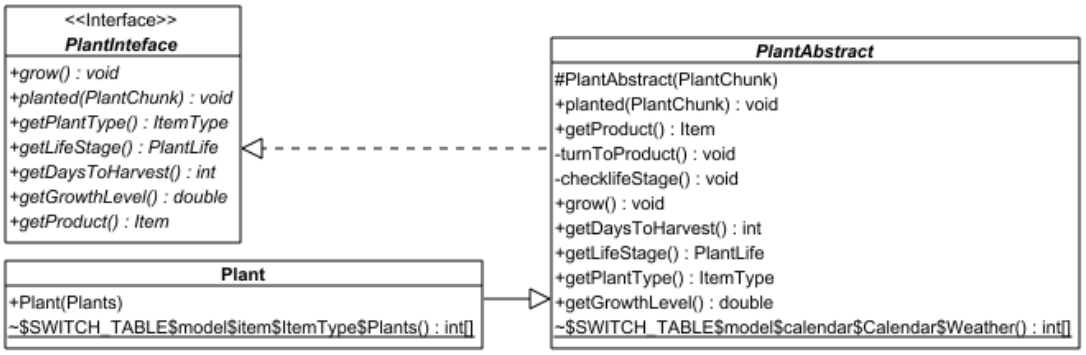

Figure 2.8: UML Diagram of the Plants

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

2.2.2.1 Person

The class `Person` and its derived classes provide a foundation for representing different individuals in the system, such as landlords and farmers, with specific properties and actions for each role.

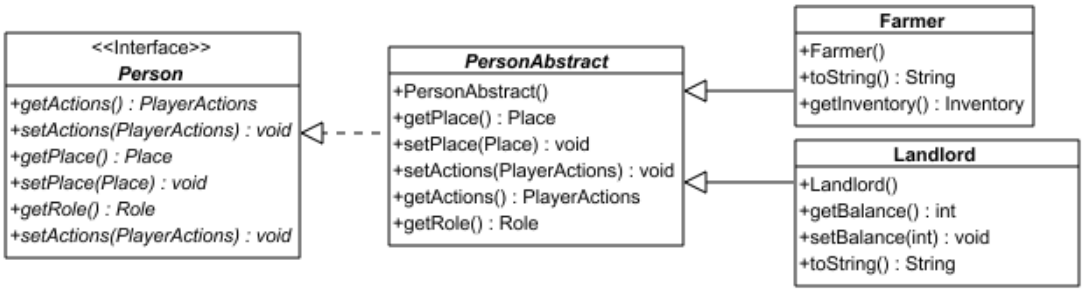

Figure 2.9: UML Diagram of the Person

2.2.2.2 Calendar

It represents a calendar system that keeps track of the day, season, and weather conditions in the game. It provides methods to increment the day, retrieve the current day, weather, and season, as well as set the weather and season.
It was implemented through the Singleton Design Pattern, to assure that every game entity had the same Calendar instance.

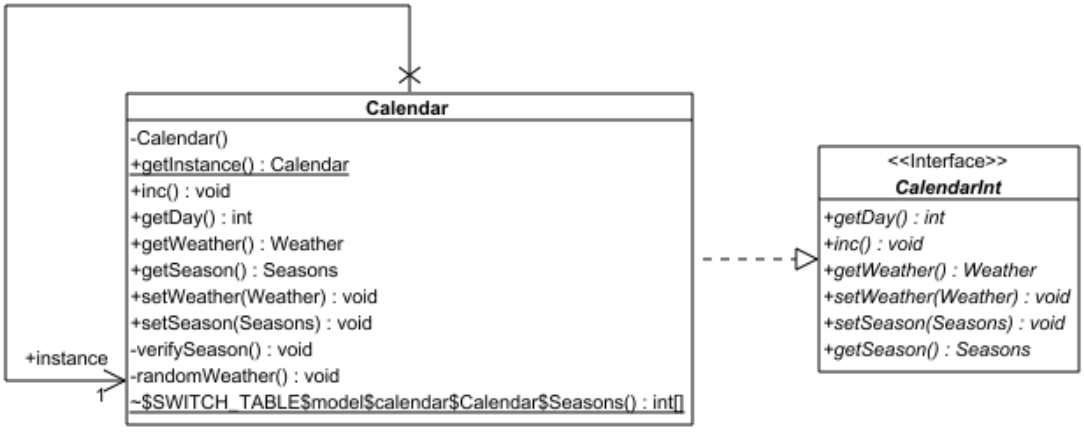

Figure 2.10: UML Diagram of the Calendar

2.2.2.3 Item

The class provides several methods to access object data:

- `getType()` returns the type of the object.
- `getStatus()` returns the status of the object.

- `getPrice()` returns the price of the object.
- `getNumber()` returns the number of objects.
- `getMaxNumber()` returns the maximum number of allowed objects.
- `setNumber()` sets the number of objects, ensuring it is not less than zero.
- `clone()` allows cloning of the object.

The class implements the `compareTo()` method for comparing Item objects based on their type. Specific sorting criteria among object types are defined. Overall, this `Item` class represents an abstract object with common attributes and methods to manage object information in the context of a game or application.

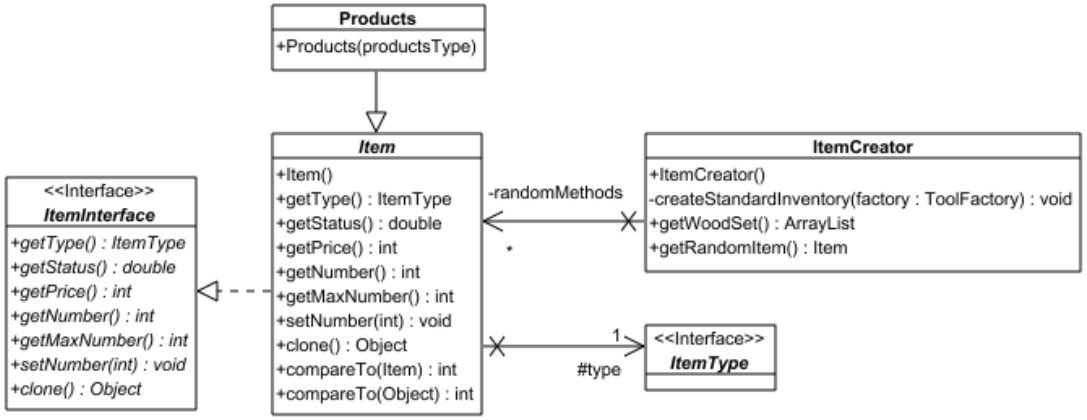

Figure 2.11: UML Diagram of the Item

Furthermore, an `ItemCreator` class was introduced to invoke the methods for item creation within the market and farmer. The class has a private method called `createStandardInventory()` that takes a ToolFactory as a parameter and adds a set of standard items to the inventory.

The `getWoodSet()` method creates a standard inventory using the `woodFactory` and returns the resulting ArrayList of items.

The `getRandomItem()` method returns a random item from the `randomMethods` array using the Random class.

Overall, this `ItemCreator` class is responsible for item creation and inventory management. It provides methods to create a set of standard items, retrieve a set of items, and obtain random items.

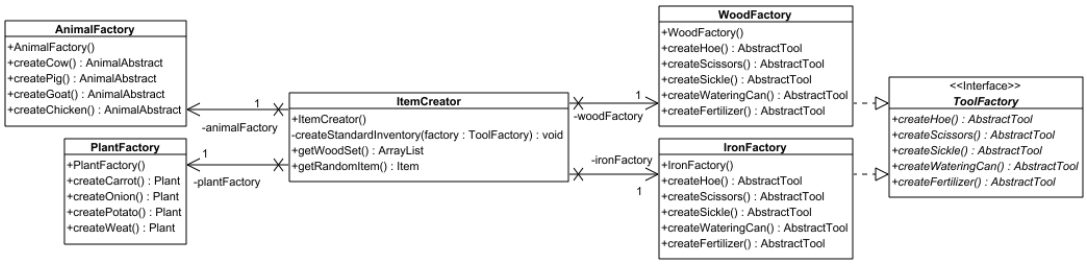

Figure 2.12: UML Diagram of the ItemCreator

For the implementation of tools, I have adopted the Abstract Factory pattern as I considered it to be the most suitable pattern since tools can be made of different materials, such as wood or iron (the durability of a tool varies depending on the material).

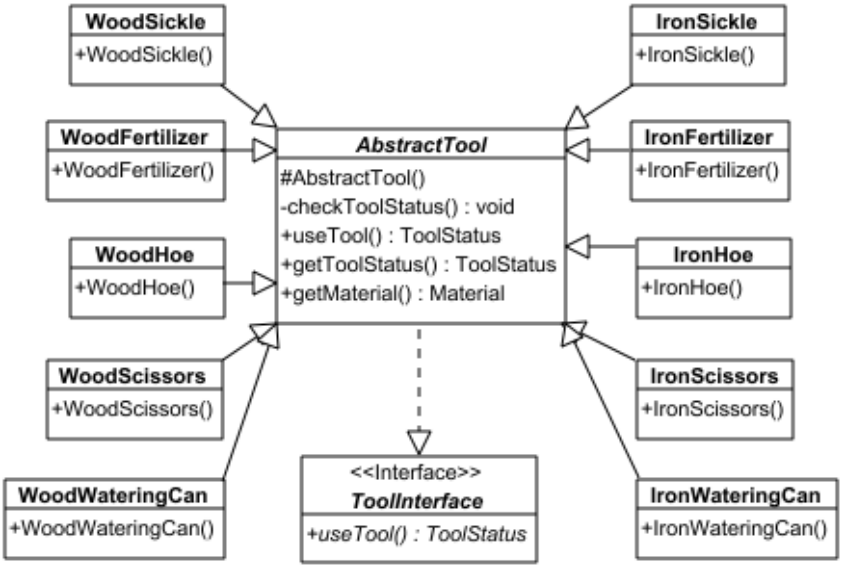

Figure 2.13: UML Diagram of the Tools

Animal:

- The abstract animal implements methods for hunger and thirst update. If the hunger or thirst levels reach a certain critical point, the animal is considered dead.
- Information about the animal can be obtained through methods such as `isAlive()` (to check if it is alive), `getHunger()` (to get the hunger level), and `getThirst()` (to get the thirst level).
- The `getProducts()` method returns a copy of the products generated by the animal, while simultaneously resetting the number of products in the animal itself. This method can be used to collect the products generated by the animal for gameplay or future use.
- The class also provides methods to check the availability of products (`areProductsAvailable()`) and update the production of products based on the animal's hunger and thirst levels (`updateProducts()`).
- The `update()` method is called to update the overall status of the animal. During the update, the hunger and thirst levels are incremented, and if the animal reaches a certain critical point, it is considered dead. Additionally, the production of products is updated based on the hunger and thirst levels.
- The `feed()` method is used to feed the animal, requiring an object of type GamePlant. The animal consumes the food, reducing the hunger level. If no food is found or the hunger level reaches the minimum allowed, appropriate exceptions can be thrown.
- The `waterAnimal()` method is used to hydrate the animal, reducing the thirst level. If the thirst level reaches the maximum allowed, an exception is thrown to indicate that the thirst level has been exceeded.

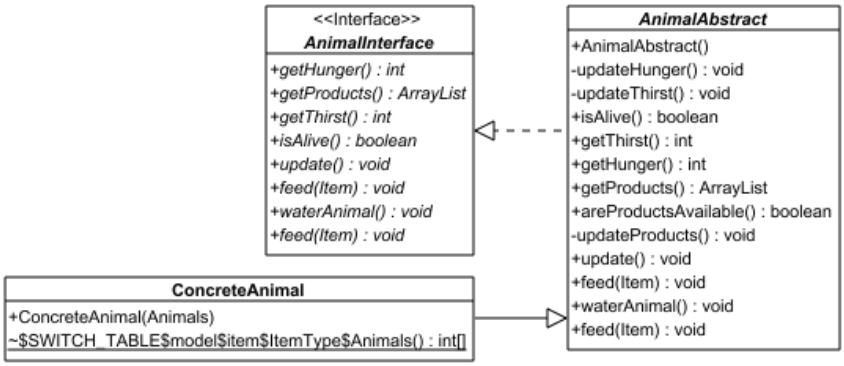

Figure 2.14: UML Diagram of the Animal

AnimalChunk

- `public AnimalLand getLand()`: Returns the `AnimalLand` object associated with this chunk.
- `public void setAnimal(AnimalAbstract animal)`: Sets the `AnimalAbstract` object associated with this chunk.
- `public AnimalAbstract getAnimal()`: Returns the `AnimalAbstract` object associated with this chunk.
- `public void removeAnimal()`: Removes the animal associated with this chunk by setting the value of the `animal` field to `null`.
- `public void update()`: Overrides the abstract `update` method of the parent class `Chunk`. Checks the state of the animal and updates the chunk's actions.
- `public void resetActions()`: Resets the actions of the chunk and allows the addition of a new animal.

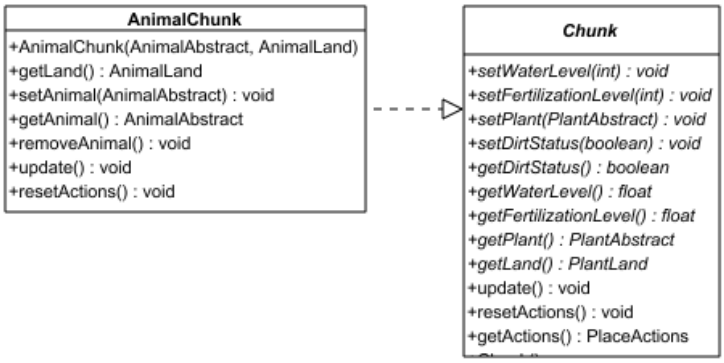

Figure 2.15: UML Diagram of the AnimalChunk

2.2.2.4 Inventory

- The `Inventory` class represents an inventory that keeps track of the items within it. It has the ability to add, remove, and search for items in the inventory.
- The `getItemInventory()` method returns a copy of the inventory of the items present, sorted based on a defined sorting criterion.
- The `addItem(Item item)` method adds an item to the inventory. It handles different cases: if the item is already present in the inventory, it updates the quantity; if the inventory is full, an `InventoryIsFullException` is thrown.
- The `removeItem(Item itemToRemove, int numItemReq)` method removes a certain number of an item from the inventory. It handles different cases: if the requested quantity matches the quantity of the existing item, the item is completely removed; if the requested quantity is less than the quantity of the existing item, the requested quantity is subtracted; if the item is not found in the inventory, a `NoItemFoundException` is thrown.
- The `searchItem(Item itemtofind, boolean accLessMax)` method searches for an item in the inventory based on its type. It returns an `Optional<Integer>` object representing the index of the found item. The `accLessMax` parameter determines whether the search should also consider the maximum quantity of the item.
- The `getItem(int numItemReq, Item itemRequest)` method retrieves a copy of a certain number of an item from the inventory. It reduces the quantity of the requested item in the main inventory. It handles different cases: if the requested quantity is greater than the available quantity, the item is completely removed; if the requested quantity is less than the available quantity, a copy with the requested quantity is created and the quantity of the existing item is reduced accordingly; if the item is not found in the inventory, a `NoItemFoundException` is thrown.

- The `setInventory(ArrayList<Item> inventory)` and `setItemInventory(int i, Item item)` methods allow setting the entire inventory or a specific item within the inventory.

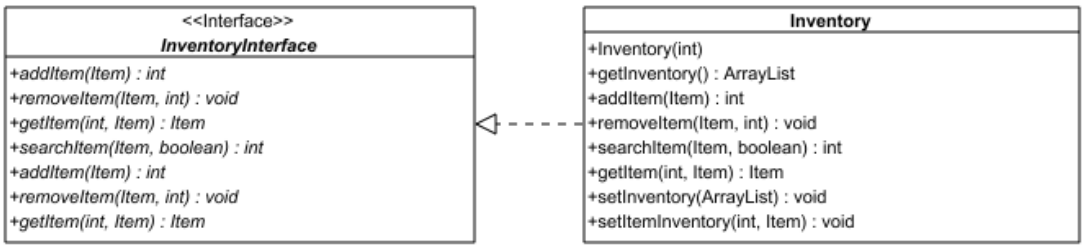

Figure 2.16: UML Diagram of the Inventory

2.2.2.5 Barn

- The getter methods return the inventory of the barn and the market.
- The method `setItemBarnInventory(Item item)` allows adding an item to the barn's inventory. Exceptions are handled in case the item is not found or the inventory is full.
- The method `updateBarn()` updates the market and the animals present in the barn. It is called to update the state of the barn in the game. During the update, the market is updated by invoking the `updateItemShop()` method of the market. The animals in the inventory are updated by calling the `update()` method of each animal. If an animal has maximum hunger levels or a minimum state, it is removed from the barn.

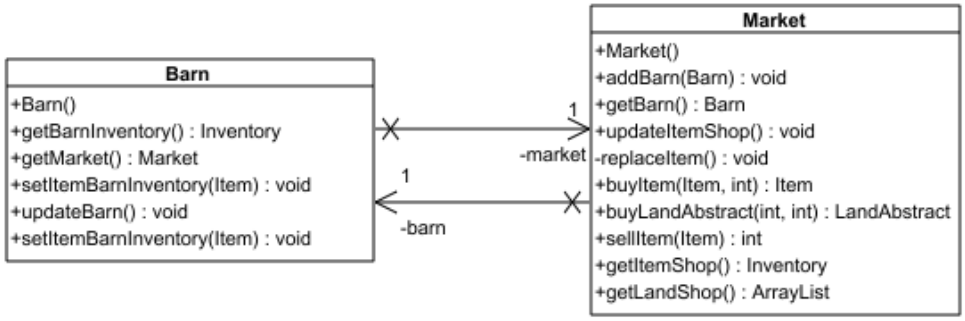

Figure 2.17: UML Diagram of the Barn

Market:

- The `addBarn(Barn b)` method sets the reference to the barn object. This allows the market to access and interact with the barn.
- The `getBarn()` method returns the barn object associated with the market.
- The `updateItemShop()` method is called to update the item shop. It checks if the current day (obtained from the calendar) is divisible by 7. If it is, it triggers the replacement of items in the shop by calling the `replaceItem()` method.
- The `replaceItem()` method replaces the items in the item shop. It first clears the current inventory of the item shop. Then, it uses the `itemCreator` to generate new random items and adds them to the shop's inventory. The items added are unique, ensuring that there are no duplicates in the shop.
- The `buyItem(Item boughtItem, int balance)` method allows players to purchase an item from the shop. It takes the item the player wants to buy and their current balance as parameters. It checks if the player's balance is sufficient to afford the item's price. If it is, it removes one instance of the item from the item shop and returns it to the player. If the balance is not enough, it throws a `NoEnoughMoneyException`.

- The `buyLandAbstract(int landIndex, int balance)` method enables players to purchase a land from the shop. It takes the index of the desired land in the land shop and the player's balance as parameters. It checks if the player's balance is enough to afford the land's price. If it is, it returns the corresponding land object from the land shop. If the balance is insufficient, it throws a `NoEnoughMoneyException`.
- The `sellItem(Item itemToSell)` method calculates the price at which an item can be sold. It takes the item to be sold as a parameter and returns half of its original price. This method is used when players sell items to the market.
- The `getItemShop()` and `getLandShop()` methods simply return the item shop inventory and the land shop, respectively, allowing other parts of the game to access and interact with them.

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()`
- `testGameBackupException()`

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 Luis Frasheri

- Management of people, including everything related to the barn, market and animals.
- Implementation of an inventory system adopted in the farmer, barn, and market.
- Development of tools used by the farmer.
- Program testing management.

3.2.3 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

- **Lambda Expressions** I implemented lambda expressions because they allow me to write cleaner and more concise code. This allows me to focus on the essence of the problem I'm solving, without having to worry about unnecessary details.
- **Streams** I used streams in combination with lambda expressions because they allow me to write cleaner and more concise code for filtering, transforming, and aggregating data operations. Streams provide a fluent and declarative way to work with data collections, enabling efficient and elegant operations such as filtering, mapping, and reducing data. The use of lambda expressions within streams allows me to specify the operations to be performed on each data element clearly and concisely, without the need for explicit iterations or writing traditional for loops.
- **Optional** The use of Optional in Inventory allows for safe and readable handling of the possibility of not finding an item in the inventory. It provides explicit control over the presence or absence of the element and avoids the need to use exceptions to represent this scenario.
- **Maven** Was a comprehensive tool that made building and testing really easy.

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, and they where overall used throught the 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.
- **Generics** where mostly used as tool to adaptively get arguments for the actions, that being the `ActionArguments` class.
- **Maven** was a comprehensive tool that made building and testing really easy.

These notes have been prepared with LaTeX. 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.