

BİLKENT UNIVERSITY

Department of Computer Engineering

CS 319 – Object Oriented Software Engineering

Spring 2018 Term Project

Instructor: Uğur Doğrusöz

Farmio (Group 2F)

Analysis Report (1st Iteration)

Group Members

Demir Topaktaş

Fuad Ahmad

Nursena Kal

Eray Şahin

Contents

[**1.** **Introduction** 4](#_Toc506631757)

[**2.** **Overview** 4](#_Toc506631758)

[**2.1.** **Gameplay** 4](#_Toc506631759)

[**2.2.** **Map** 5](#_Toc506631760)

[**2.3.** **Soil** 5](#_Toc506631761)

[**2.4.** **Store** 5](#_Toc506631762)

[**2.5.** **Seeds** 5](#_Toc506631763)

[**2.6.** **Grown Crops** 6](#_Toc506631764)

[**2.7.** **Harvesting** 6](#_Toc506631765)

[**2.8.** **Inventory** 6](#_Toc506631766)

[**2.9.** **Selling** 7](#_Toc506631767)

[**2.10.** **Farmer’s Health** 7](#_Toc506631768)

[**3.** **Functional Requirements** 7](#_Toc506631769)

[**3.1.** **Play Game** 7](#_Toc506631770)

[**3.1.1.** **Start a New Game** 7](#_Toc506631771)

[**3.1.2.** **Load Game** 7](#_Toc506631772)

[**3.2.** **Credits** 8](#_Toc506631773)

[**3.3.** **Help** 8](#_Toc506631774)

[**3.4.** **Exit Game** 8](#_Toc506631775)

[**4.** **Non-functional Requirements** 8](#_Toc506631776)

[**4.1.** **Usability** 8](#_Toc506631777)

[**4.2.** **Performance** 8](#_Toc506631778)

[**4.2.1.** **Response Time** 8](#_Toc506631779)

[**4.3.** **Supportability** 9](#_Toc506631780)

[**5.** **System Models** 9](#_Toc506631781)

[**5.1.** **Use Case Model** 9](#_Toc506631782)

[**5.2.** **Dynamic Models** 13](#_Toc506631783)

[**5.2.1.** **Sequence Diagrams** 13](#_Toc506631784)

[**5.2.2.** **State Diagram** 17](#_Toc506631785)

[**5.2.3.** **Activity Diagram** 19](#_Toc506631786)

[**5.3.** **Object and Class Model** 20](#_Toc506631787)

[**5.4.** **User Interface - Navigational Paths and Screen Mock-ups** 24](#_Toc506631788)

[**6.** **Glossary & References** 24](#_Toc506631789)

1. **Introduction**

Farmio is a basic 2D farming simulator video game we planned to develop. The main aim of this game is to manage a farmland by planting different types of seeds, taking care of plants, gathering their grown crops to either sell or eat. If the player chooses to sell the grown crops, this will facilitate in generating income, helping the player to invest in different types of seeds to attain more money. Alternatively, the player may also choose to eat the grown crops so that the health of the farmer can be kept at its maximum. The player has to balance these two actions properly since either running out of money or worsening health means the end of the game. In other words, the game will be over when the player loses all of his/her money -and there is no investment in plants to provide income- or when the farmer represented in the game loses his/her health.

Additionally, we will implement Farmio in Java programming language by Object-Oriented Programming principles and it will be a desktop application. This report contains the game overview, basic game objects, and the basic structure of the game. Besides, we also added the functional, non-functional requirements as well as the use-case, class, activity and state diagrams.

1. **Overview**
   1. **Gameplay**

The game is played using the mouse. Some actions are to be handled through right clicking to choose one of the several possible actions.

* 1. **Map**

The map initially consists of a farm house whose location is fixed (defined by the game). Surrounding the farm house are the “slots” of soil on which the player may plant and grow seeds depending on the type of the soil (see the next section *2.3. Soil* for details).

* 1. **Soil**

The soil slots can be considered in two different categories as “grass” and “pit”. Grasses are blocks of soil which are not suitable for planting. On the other hand, pits are the ones on which the player can plant and grow seeds.

* 1. **Store**

The player is able to buy seeds of different kinds from the store. The store, which is to be made available directly on the game screen, displays the available seeds and their respective prices as small icons.

Note that, at the beginning of a new game, the player is provided with some initial money to buy some seeds to begin planting. Except this initial money, the actual income is supposed to be generated through growing crops and selling them.

* 1. **Seeds**

The game represents strawberry, corn and sunflower seeds available for purchase at the store. These seeds differ by their growing times and prices. Having planted seeds, as long as they are watered *once* using the “watering can” tool, they grow continually and form grown crops. On the other hand, when they are not watered for a specific time limit, they will spoil, making the farm slot in which they were sown available for planting again.

* 1. **Grown Crops**

As already mentioned above, seeds produce grown crops (also called “food” throughout the report) when treated properly. These crops need to be harvested as soon as possible, otherwise they will spoil (just like the seeds that are not watered), which is indicated by the change of the color of the soil. Again, rotten food makes the farm slot available for planting different seeds (as if it has never been cultivated before).

* 1. **Harvesting**

When the seeds are grown fully and have produced grown crops, which is indicated by slots’ attaining new icons, the player is expected to harvest them before they spoil. To harvest the grown crops, the player simply clicks on these new icons.

* 1. **Inventory**

The purchased seeds and collected crops appear directly on the inventory, which also is continuously accessible on the game screen for ease of use. The player can switch back and forth between both the purchased seeds to plant and also between gathered crops to eat or sell.

* 1. **Selling**

To sell the gathered crops and generate income, the player may right click on the harvested items on the inventory to select “Sell”. Note that each crop will yield a different amount of money, depending on the kind of the seed from which it is grown.

* 1. **Farmer’s Health**

As one of the main objectives of the game, in addition to maintaining money, the player needs to keep the health of the farmer highest so as to keep the game continuing. This involves the player’s letting the farmer eat some of the gathered crops through using the inventory. Note that there will *not* be a farmer represented by a person moving around the farmland. Instead, it will be conceptually referred by its health and money (that is, to be displayed within a box on the screen).

1. **Functional Requirements**
   1. **Play Game**
      1. **Start a New Game**

The player is allowed to launch a new game anytime without needing to override any previous progress. In other words, the player may maintain multiple “save files” with each having a possibly different progress.

* + 1. **Load Game**

Since the player is granted access to multiple save files as mentioned above, this will allow the player to choose and load one of the previously saved games.

* 1. **Credits**

This screen displays some information about us as the developers of the game.

* 1. **Help**

The player can access this screen to learn more about the game controls as well as the objective of the game.

* 1. **Exit Game**

The player may exit the game through this option.

1. **Non-functional Requirements**
   1. **Usability**

The game should represent an intuitively understandable user interface. More specifically, the positions of the store icon, inventory and the farmer icon should be easy to identify and convenient to interact with.

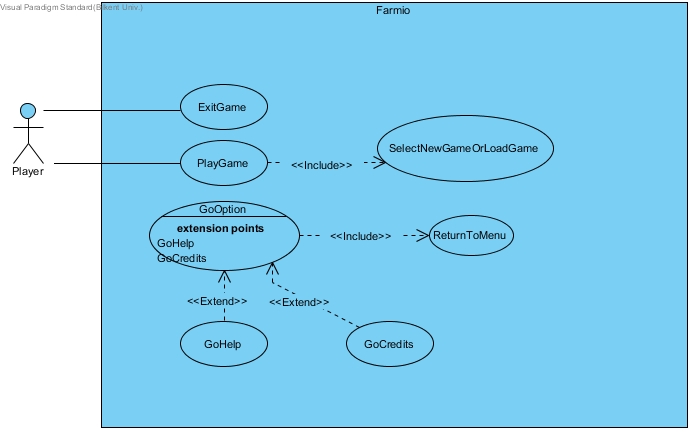
* 1. **Performance**
     1. **Response Time**

The game should respond to the input provided by the user as quickly as possible. Primarily, the images need to be updated instantly to indicate the changes the player has made, also providing a smoother gameplay.

* 1. **Supportability**

Since the player can start a new game anytime in addition to the previously saved ones (also mentioned in *1.1.1 Start a New Game*), the generation of this new game should not cause any changes to the previously saved games. That is, the game system should handle all save files separately, independent of each other.

1. **System Models**
   1. **Use Case Model**



Use Case #2

**Use case name:** GoHelp

**Participating actors:** Player

**Entry condition:** Player firstly opens the game and goes to Options page.

**Exit condition:** Player returns to main menu.

**Main Flow of Events:**

1. “Help” button is chosen by Player.
2. Player gets informed about the instructions of the game.
3. Player returns to the Option menu.

Use Case #1

**Use case name:** PlayGame

**Participating actors:** Player

**Entry condition:** Player has to open main Menu and has to select “Play” button

**Exit condition:**

1. Player has to lose game by finishing his money, OR
2. Player has to exit the game by clicking the exit button.

**Main Flow of Events:**

1. Player clicks the "Play Game" Button
2. System comes up with selection page that asks if player wants to load game or start a new game
3. Player plays the game and returns to main menu at the time player wants.
4. Player returns to the main menu of the game.

**Alternative Flow of Event:**

1. Player loses all of his/her money and system comes up with pop-up message that “Game Over” and game returns to Main Menu
2. Player can exit the game any time.

Use Case #4

**Use case name:** GoCredits

**Participating actors:** Player

**Entry condition:** Player firstly opens the game and goes to Options page.

**Exit condition:** Player returns to main menu.

**Main Flow of Events:**

1. “Credits” button is chosen by Player.
2. Player gets informed about the creators of the game.
3. Player returns to the Option menu.

Use Case #3

**Use case name:** ReturnToMenu

**Participating actors:** Player

**Entry condition:** Player firstly opens the game and goes to Options page.

**Exit condition:** Player returns to main menu.

**Main Flow of Events:**

1. Player chooses "Return Menu" from Options menu.
2. Player returns to main menu.

Use Case #6

**Use case name:** ExitGame

**Participating actors:** Player

**Entry condition:** Player is on main menu.

**Exit condition:** Program is closed.

**Main Flow of Events:**

1. Player clicks "Exit" button from main menu.
2. Program is closed.

Use Case #5

**Use case name:** SelectNewOrLoadGame

**Participating actors:** Player

**Entry condition:** Player chose “Play” button from main menu.

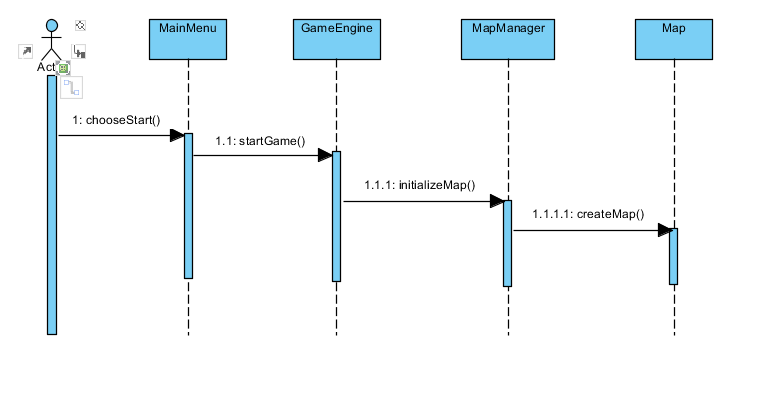
**Exit condition:** Player returns to main menu.

**Main Flow of Events:**

1. Player clicks "Play Game" button from main menu.
2. Player chooses to play new game or load game.
3. Player returns to the main menu of the game after playing game.

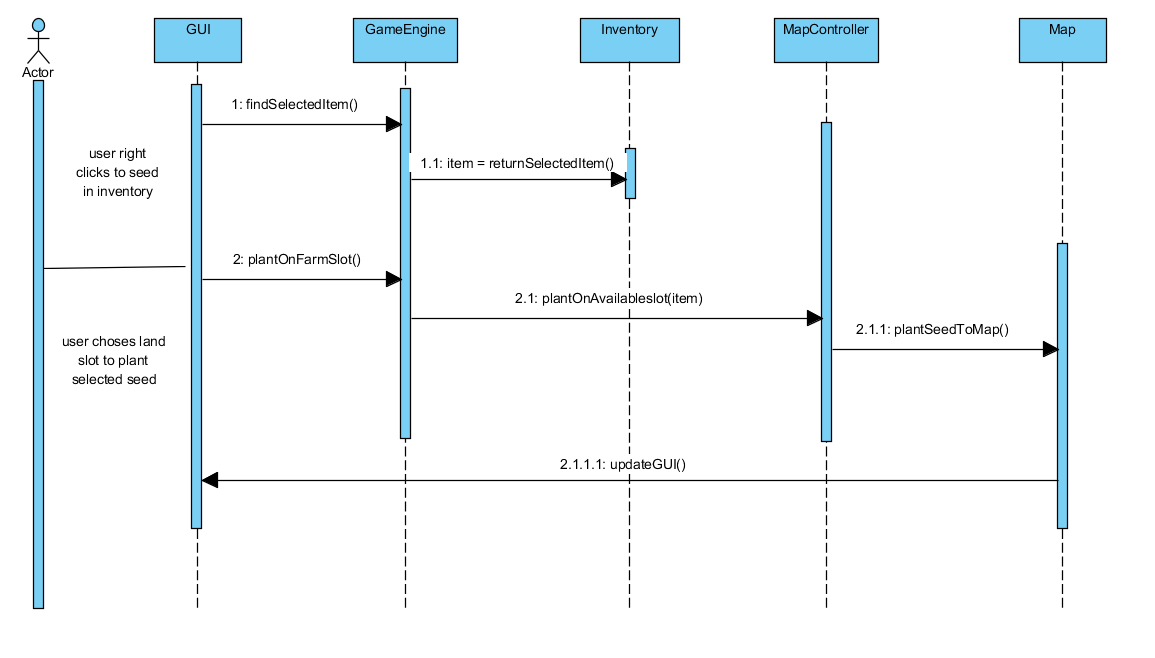
**Alternative Flow of Event:**

* Player returns to main menu without playing the game.
  1. **Dynamic Models**
     1. **Sequence Diagrams**



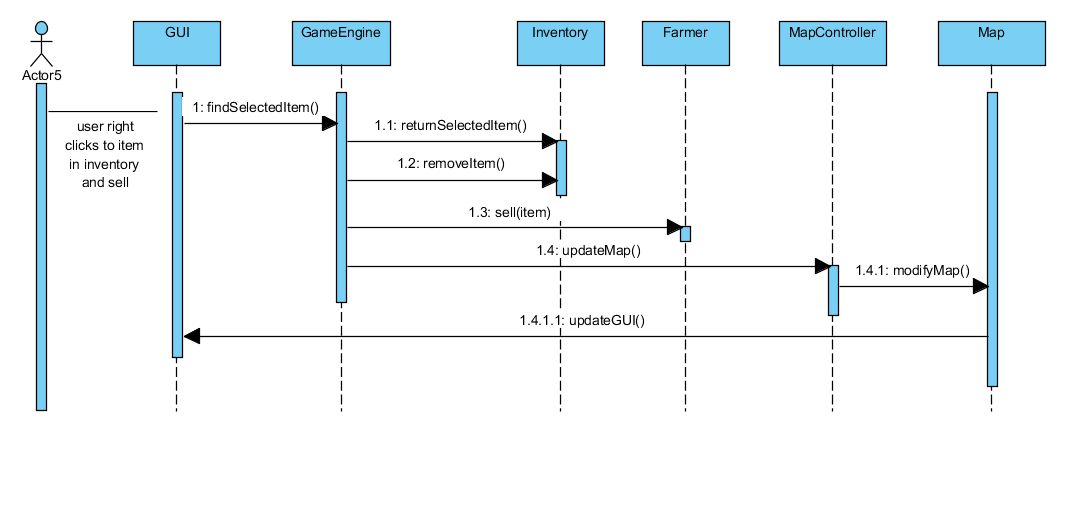
**Scenario:** Starting the Game

The player is, firstly, shown the *MainMenu*. Then, after choosing either to load an existing game or start a new one, the *GameEngine* starts the game. *MapManager* initializes the map respective to the choice of the player (either commands a new map to be created or asks for one of the previously saved games’ map to be loaded). Finally, *Map* is to be created according to the *MapManager*’s instructions.



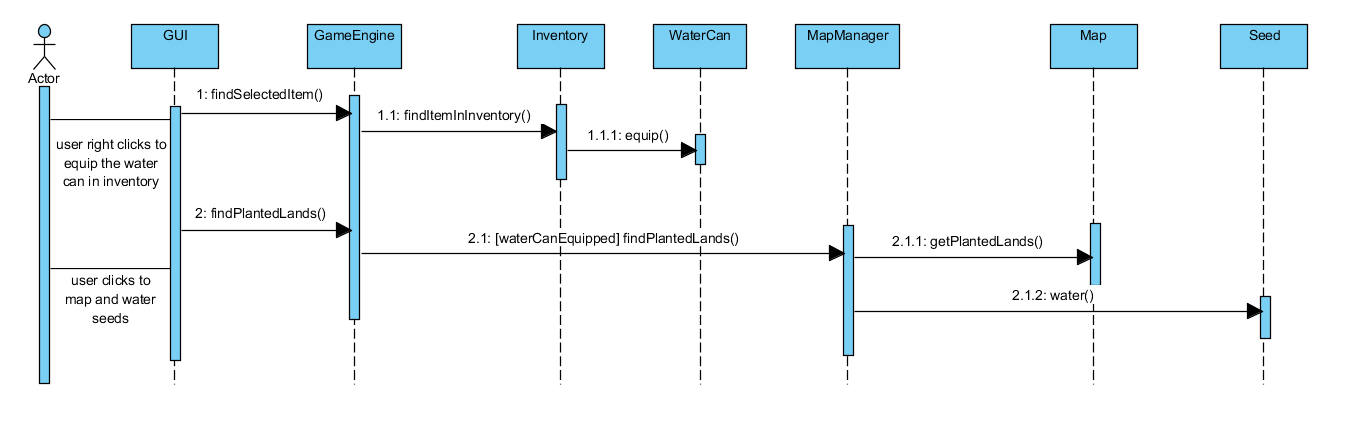
**Scenario:** Planting Seeds

To plant an already-purchased seed, the player right clicks on one of the available seeds in the inventory. *GameEngine* detects which item is chosen by the player. Then, the *Inventory* returns the instance of the selected item. The player is now expected to choose a land slot on which the chosen seed is to be planted. After the player has clicked on a land slot, *MapController* receives the instance returned by the *Inventory*, checks if the player’s choice of a land slot is suitable for planting. If so, the seed is planted to the slot through *Map*’s *plantSeedToMap*() function. Finally, the screen is updated by *GUI*.



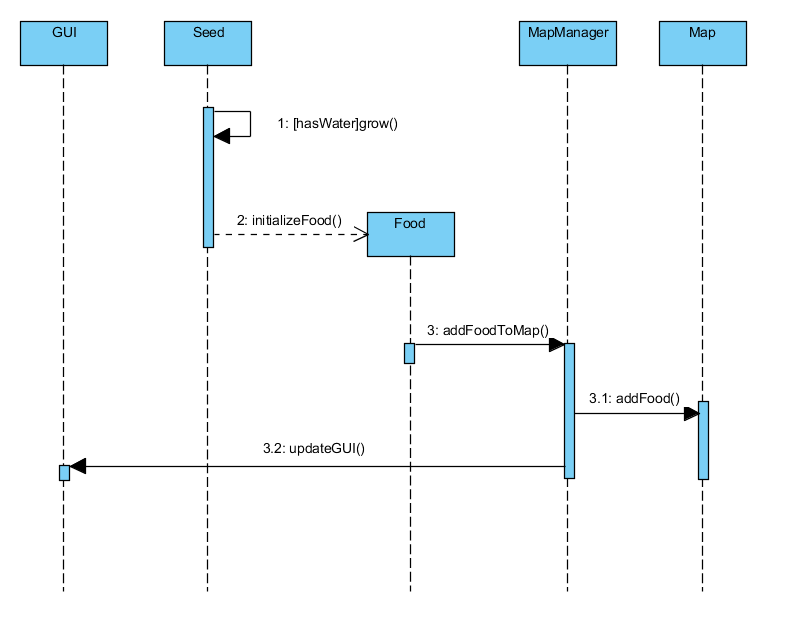
**Scenario:** Selling a Grown Crop

To sell a grown crop (i.e. food), the player right clicks on one of the collected crops in the inventory. As in the above case, *GameEngine* detects the clicked item. Again, *Inventory* returns the instance of the chosen item and removes it. The returned item is received by the *Farmer* instance to be sold. *MapController* is then expected to indicate the change in the inventory (the sold food is to be disappeared) and call *GUI*’s respective function to update the screen.



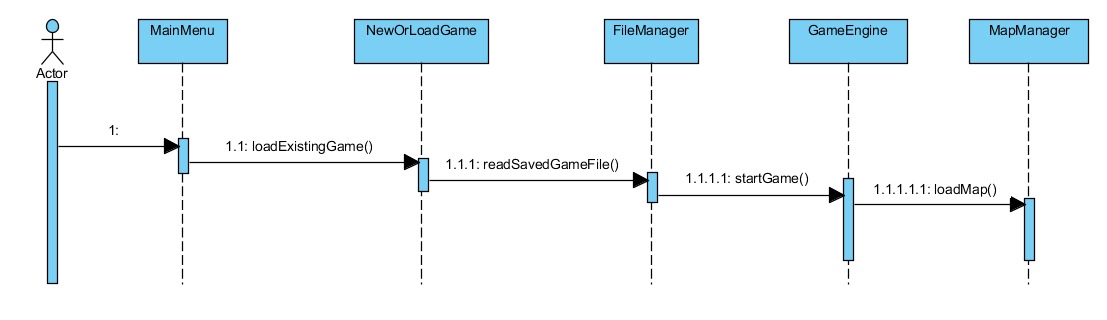
**Scenario:** Watering Seeds

The player right clicks on the watering can available in the inventory. As usual, *GameEngine* detects that an item, in this case the watering can, is selected. The instance of the watering can is to be returned from *Inventory* and is equipped. Then, the player is expected to click on land slots bearing seeds. Having ensured the player has already equipped the watering can and clicked on a slot, the *GameEngine* (and *MapManager* in turn) controls whether the clicked slot is cultivated. If so, the seeds on that slot are watered.



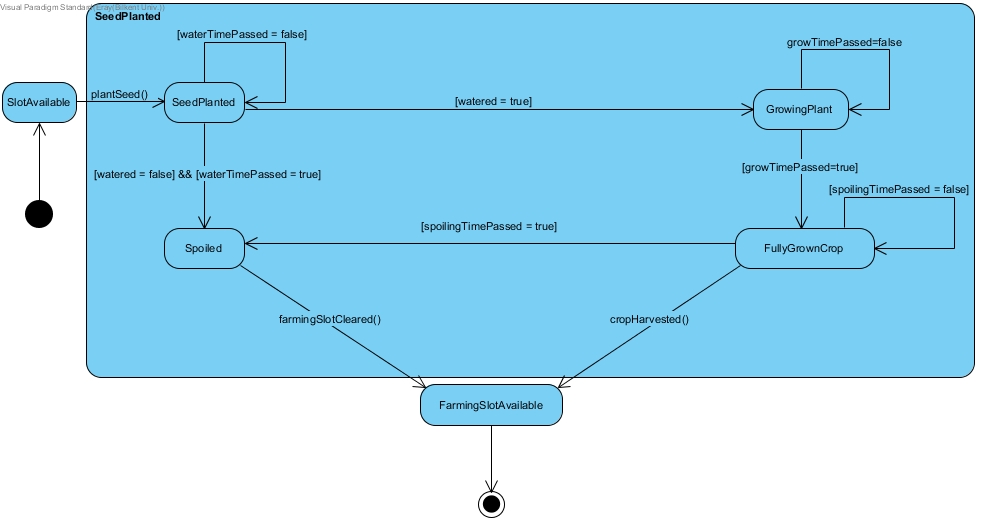
**Scenario (Watering Seeds Continued):** Producing Grown Crops / Food

The final diagram above, despite not representing a complete scenario on its own, is a follow-up of the watering process. In other words, after watering the seeds on a specific land slot, the seeds grow continually. When they reach their *FullyGrownCrop* state, which is illustrated in detail in the state diagram below, they construct *Food* objects. These instances are added to the inventory and displayed on the screen by the respective update just as in the previous scenarios.

**Scenario:** Loading Saved Game

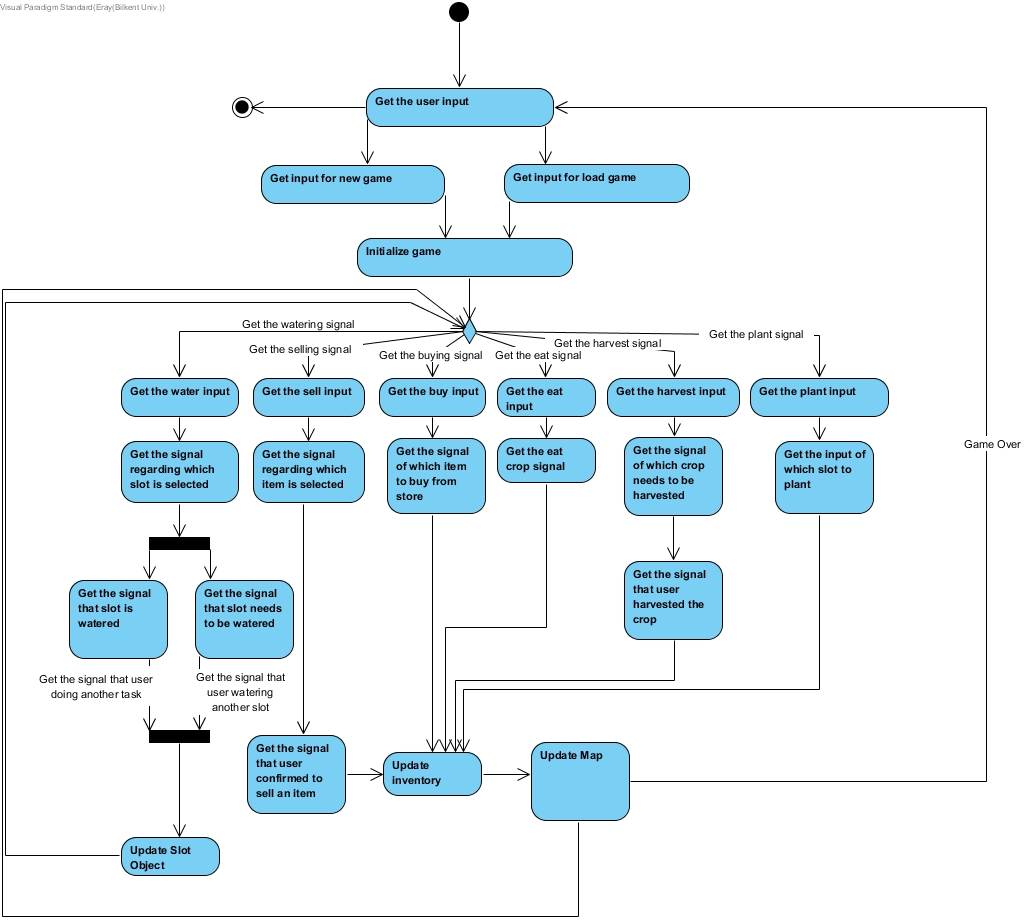
The player chooses to load an existing game. After a call to the function *loadExistingGame()* from *NewOrLoadGame* object, *FileManager* reads the saved game that the player wants to load from the text file. Using the information available there, *GameEngine* starts the game and *MapManager* loads its respective map.

* + 1. **State Diagram**



The above diagram illustrates the states through which seeds proceed. Initially, after having been planted on an available slot, seeds need to be watered within a specified time (determined by *waterTimePassed*). If these sown seeds are not watered within that time, they are spoiled. On the other hand, *watered* seeds form growing plants. They stay in this state (keep growing) until they form their *FullyGrownCrop*. Just like seeds, *FullyGrownCrops* are spoiled when they are not harvested within a time limit (to be clear, *spoilingTimePassed* determines grown crops’ spoiling time whereas *waterTimePassed* defines that of the seeds). In all cases of spoilage -whether it is a seed or food that spoils- or gathering of the grown products, the land slot becomes available for planting new seeds. In other words, spoilage and harvesting cause land slots to be “reset”, making them available for planting from scratch as also mentioned in the overview.

* + 1. **Activity Diagram**

****

The activity diagram above demonstrates the system behavior, mostly focusing on the playing mechanism. The game first receives the player’s input to understand whether it is a new game that is to be initialized or an existing one to be loaded. Then, after starting the game in both cases, it expects the player to perform one of the following actions: buy, plant or water seeds; sell or eat food; harvest crops.

If the player is to water seeds, the game determines which slot is to be watered. While completing the watering process for a specific slot, the game simultaneously keeps track of other slots which the player may also tend to water. Eventually, the slots (their respective instances) are updated.

When the system detects that the player is to sell food, the selected item is determined. Having recognized that the player is confirmed to sell that item, the inventory is updated with recent changes reflected on the screen.

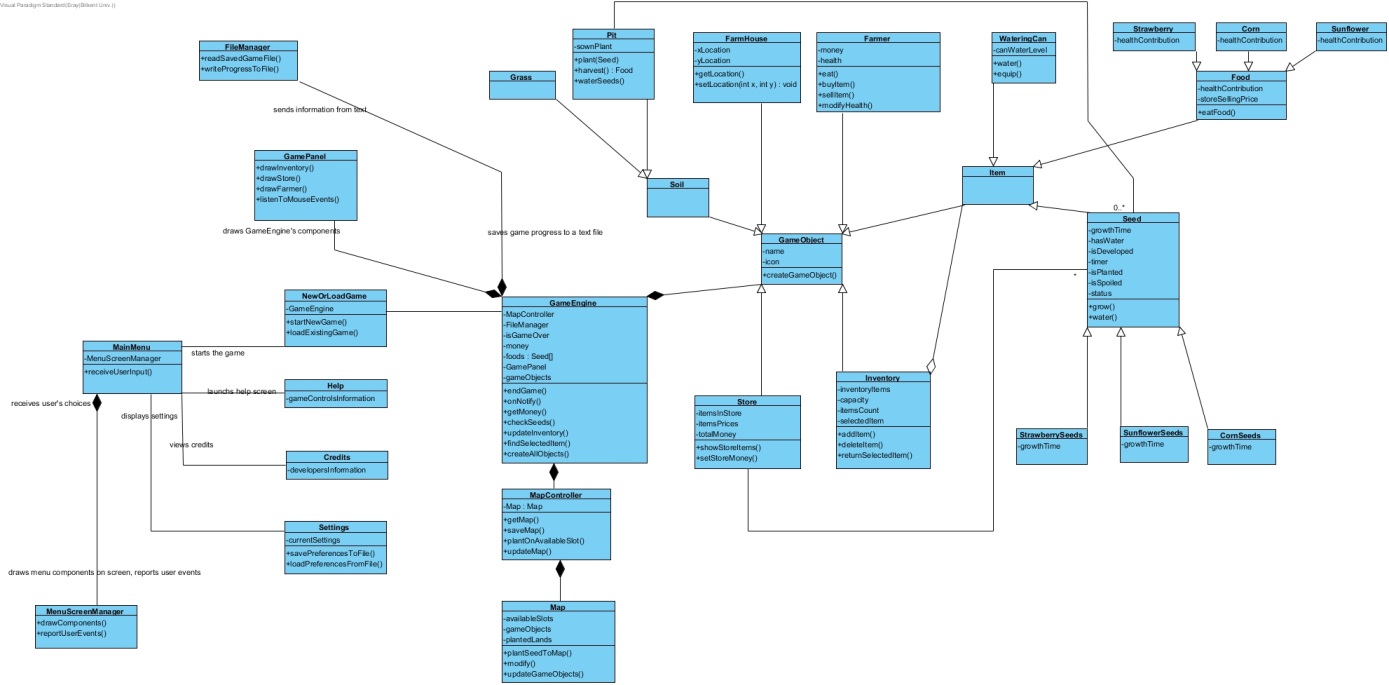
Recognizing the signal to buy involves the identification of the item to be bought. Then, both the inventory and the screen are updated with respect to the purchased item(s).

Having received the signal to eat food, the game identifies the crop to be eaten by the farmer. The inventory gets updated to indicate the decrease in the amount of the eaten item. The changes, as usual, are made available on the screen.

When the system perceives that the player is to harvest items, the crop to be harvested is determined. After realizing that the crop is harvested, the inventory receives (displays) the gathered crop.

Finally, the system may receive the player’s intention to plant seeds. In this case, the slot to be cultivated is determined, both the inventory (to indicate seed consumption) and the farmland (with sown soil slot icons) are updated.

* 1. **Object and Class Model**



Although tentative, the class diagram above illustrates the classes we are most likely to use. Following are the short descriptions for these classes:

*MainMenu*: This class is responsible for menu-related actions. As *MenuScreenManager* draws available options on the screen and reports user’s events to this class, *MainMenu* may choose to start the game (new or an existing one), launch help screen, view credits or display settings.

*NewOrLoadGame*: This class, as the name suggests, is in charge of starting a new or an existing game. It owns a *GameEngine* object, which is later to be defined from scratch (new game) or reloaded according to the description saved in a text file (saved game).

*GameEngine*: This can be considered the core class of the game. It contains a game loop. It has variety of different usages such as initializing the game, checking whether the game is ended and communicating with *GameObjects* and *MapController*.

*FileManager:* Reading the saved properties of a game from a text file as well as recording new data to the same (or even new) text file(s) are the responsibilities of this class.

*GamePanel:* Having obtained the properties of an existing game (or defined those of a new one), *GamePanel* draws the inventory, store and farmer boxes on the screen.

*MapController:* Controls the *Map* object and acts as a layer between *Map* and *GameEngine*. It loads the map and frequently checks it for any change. It communicates with *GameEngine* and gets from or sends messages to it.

*Map*: This class holds *GameObjects* and their locations in the map. It also keeps track of available and planted lands in the *Map*. It works with *GUI* to draw these objects.

*GameObject:*  As the game requires items to be visualized on the screen, most of the objects are expected to have a name and icon. Therefore, this class is the parent of all such classes.

*Store:* This class will be used to hold the *Item* objects available for purchase. Therefore, it proves as an intermediate platform for the player to interact with *Item* instances.

*Inventory:* The *Inventory* class is another medium for the player to interact with *Item* instances. The purchased seeds and gathered food, both of which are collected under the superclass of *Item*, are to be maintained in this class.

*Item:* This isa superclass representing a general concept of items. More specifically, *WateringCan*, *Seed* and *Food* classes are children of this class such that the interactions of *Store* and *Inventory* classes with these subclasses are handled more easily.

*WateringCan:* This is one of the children of the *Item* class as mentioned above. *WateringCan* class simply represents a watering can that can be equipped, has a concrete water level which decreases as being used on soil slots bearing seeds.

*Seed:* *Seed* is the superclass of *StrawberrySeeds*, *SunflowerSeeds* and *CornSeeds*. Seeds have varying growth times (depending on the type of the seed), which are to be assigned different values in the subclasses. They also have attributes representing their water condition, state of development and spoilage (in addition to the flag holding if being planted at all).

*Food:* As referred multiple times throughout this report, seeds produce grown crops and these are considered as the children of the *Food* class. Therefore, the *Food* class has three children (as there are three types of seed): *Strawberry*, *Corn* and *Sunflower*. Note that each of these children has different contributions to the health of the farmer when eaten, in addition to their selling prices’ being differentl.

*Farmer:* The farmer, which is to be represented as a static image on the screen, has attributes of money and health. When the farmer buys / sells items, the money will be increased / decreased. On the other hand, in case of food consumption, the health will be increased.

*FarmHouse:* There will be a farmhouse with a fixed location at the beginning of a new game (of course, it will remain there when the same game is continued). Therefore, this class has location-related attributes and operations.

*Soil:* As the superclass of *Grass* and *Pit*, this class just appears as a conceptual entity. The main focus will be on the *Pit* class that can hold sown seeds. Therefore, *Pit* class has an intermediate role in the interactions of the player with *Seed* instances. In other words, seeds may be accessed through *Pit* objects to perform further work.

*Help:* This is a simple class to indicate the *Help* screen. The information of game controls is stored within this class (in addition to possible visual properties that are not shown).

*Credits:* Similar to the *Help* class, a description about us is given through this class. Again, properties related with the user-interface are not shown in the diagram.

*Settings:* The player may access and modify the settings by interacting with this class. The preferences are both read from and written to a text file.

* 1. **User Interface - Navigational Paths and Screen Mock-ups**