Bilkent University

Department of Computer Engineering

CS 319 Term Project

Farmio

Group 2.F

Design Report (Iteration 2)

Demir Topaktaş, Fuad Ahmad, Nursena Kal, Eray Şahin

Supervisor: Uğur Doğrusöz

Design Report (Iteration 2)
April 17, 2018

Contents

Design Report (Iteration 2)

Farmio

1. Introduction

1.1. Purpose of the System

Farmio is a basic 2D farming simulator video game. The game, which will be a desktop application, is to be implemented in Java using Object-Oriented Programming principles. The main aim of the game is to manage a farmland by planting different types of seeds, taking care of them (by watering) and harvesting their grown crops. The player may either sell or eat the collected crops. When the player chooses to sell them, this will facilitate in generating income and help the player invest in different types of plants. Alternatively, the player may also choose to eat the collected food so that the health of the farmer, which normally decreases by time, can be kept at its maximum. The player has to balance these two actions (selling and eating) properly as either running out of money or worsening health means the end of the game. In other words, the game ends when the player spends all of the farmer’s money -and there is no investment in plants to provide income- or when the farmer loses his/her health completely.

In addition to those described above, some new features were added for the second iteration. Firstly, trees, which share common properties with seeds, were added to the game. Further, two power-ups called “genetically-modified crop” (GMC) and “fertilizer” are introduced. Unlike these power-ups which may have side-effects, the concept of “rain” that helps plants to get watered naturally is also something unique for the second iteration, too.

1.2. Design Goals

Before decomposing the system into pieces, it is crucial to mention our design goals regarding the system. In this respect, many of our design goals inherit from non-functional requirements of our system that are provided in the analysis stage.

End User Criteria:

Simulation of real world: Since our system is a game, it should provide good entertainment for the player. In order to provide the entertainment player should not have a difficulty in using our system. In this respect, system will provide player friendly interfaces for menus, by which player will easily find desired operations, navigate through menus and perform the desired operations. While a player is "farming" , speed will be important. If the seeds grow too slow, player might be bored or if too fast again the player might feel that what he has done is not satisfying. Also, it will be important to have good graphics to keep player interested. It is determined that our system will perform actions according to mouse input from the user, like clicking buttons, moving around the farm. This makes it easy to use the system from the point of the player.

Basic and applicable: The player is not ought to have knowledge about how the game is played (like how to plant a seed, how to water and plant a seed). Also, there will be a little instruction set in the beginning of the game so the player can learn hands on. Simple logic and observation of real farming will also help user to understand how to play game.

Maintenance Criteria:

Modifiability: In our system it needs to be easy to modify the existing functionalities of the system to let it accommodate new ones. More specifically, having completed the first iteration, to realize the changes brought by the second iteration, we aim to make the code "modifiable". By mentioning "modifiability", rather than any future development concerns, we are concerned with an implementation which minimizes changes to the existing functions when a new function/change is to be handled. That's why we are aiming for "modifiable" (independent) code - in case anything goes wrong and this requires changes modifications to existing functions. This can be thought to be more of a precaution to prevent us from rewriting the existing code again in case we fail to integrate new capabilities.

Performance Criteria:

Response Time: For the games, it is vital that users' requests should be responded immediately in order not to distract the player's interest and entertainment. Our system will respond player's actions almost immediate, while also displaying animations, effects smoothly for enthusiasm.

Trade-offs:

Ease Of Use and Learning vs. Functionality:

In games industry, ease of use and ease of learning are important aspects for making the game more appealing. On the other hand, customers expect enough features and functionalities to get entertained in the game. Thus, these two topics conflicts with each other. We try to balance them (give them equal importance) by providing an intuitive experience and basic functionalities.

Response Time vs. Modifiability

Modifiability requires extra calls such that the function implementations can be less dependent on each other (regarding modifications to one's affecting the others). Yet, such unnecessary calls will lead to a slower execution which, in turn, reduces the response time. In this case, we will pay more attention to response time as it affects the gaming experience of the player.

2. High-Level Software Architecture

2.1. Subsystem decomposition

To decompose the system into subsystems (or components), we preferred to use Model-View-Controller (MVC) as an architectural style. Since MVC is a favorable architectural style for user-interactive software development in general, as in games, we judged that it may prove an appropriate candidate for our project.

We have come up with specific associations between components as MVC requires (see the diagram below). Especially, we can see these associations in our system where "Controller" modifies "Model", and "View" displays "Model". MVC seems likely to help us accomplish highly coherent yet low-coupled subsystems and, in turn, a less complex system (as intended through employing an architectural style). Thus, MVC is an appropriate architectural style for our system that we are going to implement.

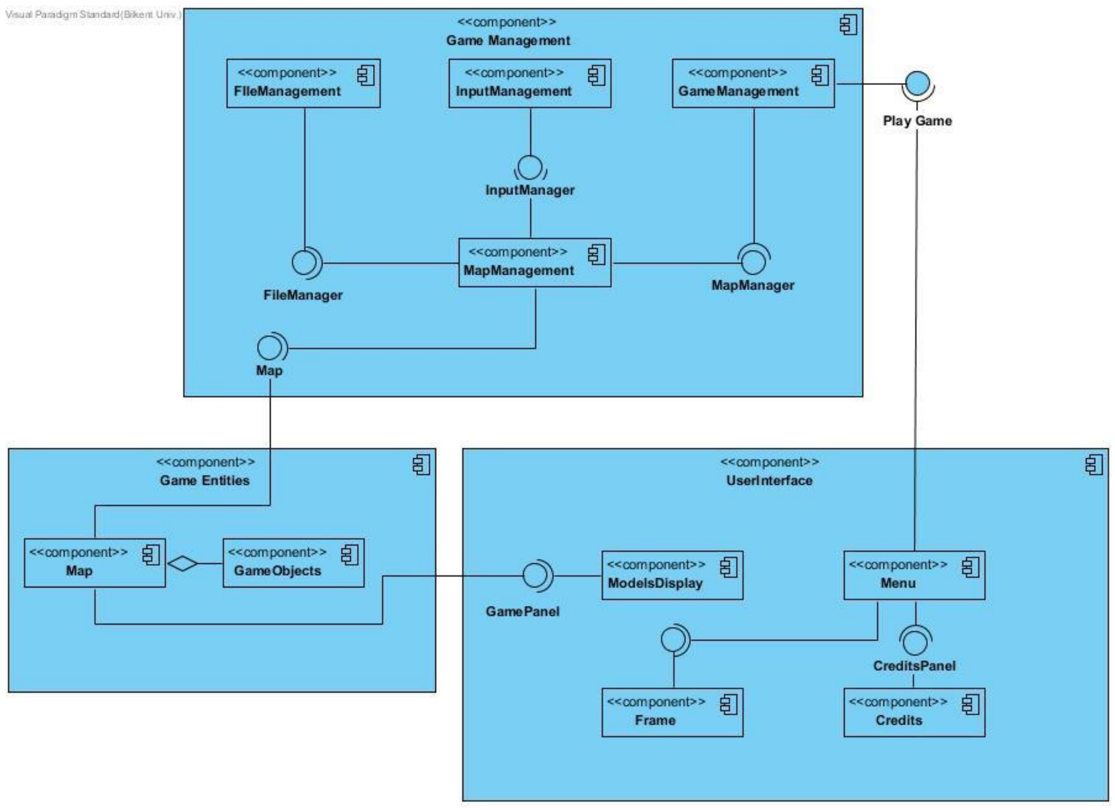

Visual Paradigm Standard(Bilkent Univ.)
<<component>> Game Management
<<component>> FIleManagement
<<component>> InputManagement
<<component>> GameManagement
Play Game
InputManager
<<component>> MapManagement
FileManager
MapManager
Map
<<component>> Game Entities
<<component>> Map
<<component>> GameObjects
<<component>> UserInterface
<<component>> ModelsDisplay
<<component>> Menu
GamePanel
CreditsPanel
<<component>> Frame
<<component>> Credits

Figure 1: Subsystem Decomposition

2.2. Hardware/software mapping

Regarding the software aspects, Farmio is to be implemented in Java, using Java Development Kit (JDK) 8. Therefore, as long as Java Runtime Environment is installed, the game can be played on Windows, Mac and Linux platforms.

To play the game, in terms of the hardware requirements, the player will need an ordinary computer with a mouse/trackpad and screen.

2.3. Persistent data management

Farmio involves data storage in a filesystem rather than a database. The reason for choosing "filesystem" approach is that there will not be any concurrent writers and, thus, a filesystem proves useful for the relatively trivial requirements of the game. For this reason, the data will be stored in text files on the hard drive. Whenever the player makes some progress, planting new seeds or harvesting grown crops for instance, these text files will be updated. Thus, in the next run, objects can be initialized properly by reading these saved information.

Additionally, as the player may maintain multiple games with each having a different progress, there will be separate text files for different games. Depending on the player's choice of whether to start a new game or load an existing one, the text file(s) to read from and/or write to will vary.

2.4. Access control and security

Farmio does not have any network-related uses, meaning that there will not be any outside users other than the user(s) of the computer in which it is stored. As a result, no measurement regarding access control and security needs to be taken.

2.5. Boundary conditions

Initialization

The game is to be launched using a ".jar" file. It will not require any prior installation.

Termination

To terminate the game, the player is supposed to switch either to the main menu or to the pause menu by pressing Esc on the keyboard during the game. Then, after choosing whether to save the progress or not, the player will be able to quit the game.

Note that since the user's decision regarding whether to save the progress cannot be determined in advance, any changes made by the user are constantly saved to a .txt *copy* of the current game. If the user eventually chooses to save the progress, this copy gets to overwrite the original "save file" of the current game. On the other hand, if the user prefers to discard the changes, the copy file is ignored while the original save file remains the same as before.

Failure

To prevent the unfortunate effects of any possibly failures, reading / writing the text files should be given the proper priority. In other words, in case the game fails and needs to be started again, the text files should have already been updated very recently. Therefore, when the game crashes somehow, in the next run, the most recently saved configuration of the game can be loaded.

Note that this will not cause any conflict regarding user-determined game saving. That is, regardless of the player's choice, all the changes are constantly saved in text files. Then, when the player wants to exit the game, these information is either to be deleted or kept as a "saved game" for following use.

3. Subsystem Services

As already mentioned in Section 2.1 (Subsystem Decomposition), the architectural style we have chosen is MVC. The three main components of MVC - model, view and controller- perform the following:

3.1. UserInterface Subsystem

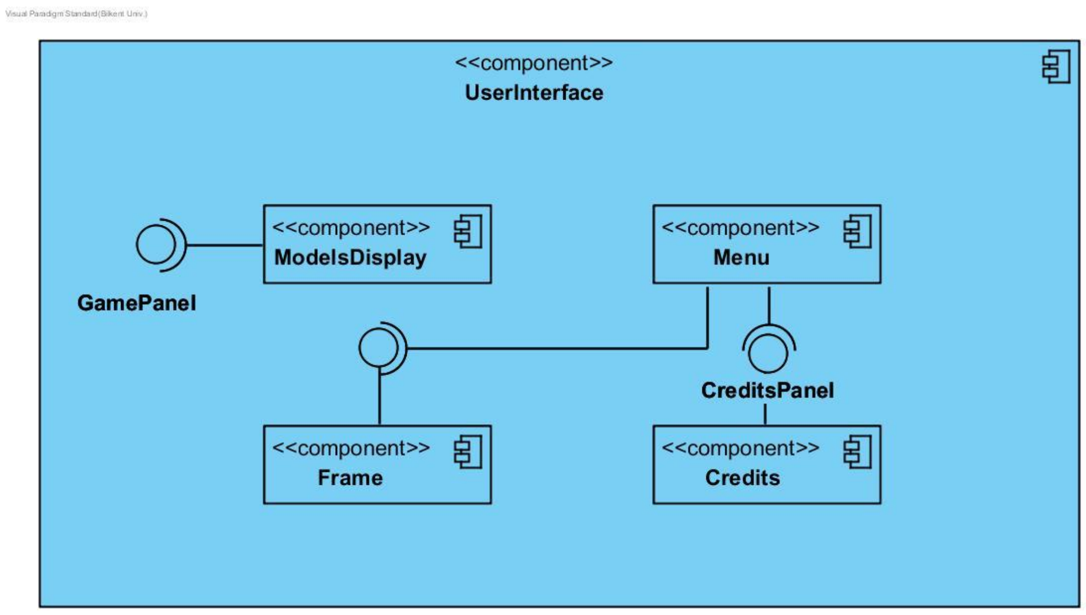


Figure 2: User Interface Subsystem

This is the “View” correspondent of MVC. Additionally, “User Interface” component has its own subcomponents including “Frame”, “Credits”, “Menu” and “ModelsDisplay”. “ModelsDisplay” is for the association between “Model” and “View” and it displays objects of the “Model”. Besides, “Menu” subcomponent is responsible for visual aspects of the menus. It receives data from “Credits” and will be displayed by the “Frame” subcomponent.

3.2. GameManagement Subsystem

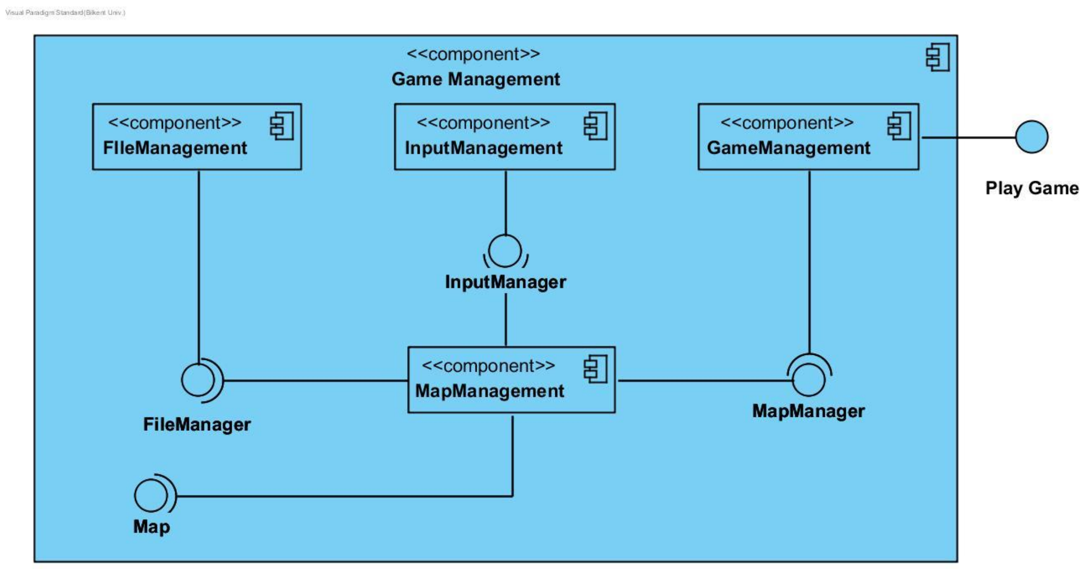


Figure 3: Game Management Subsystem

Game Management: This is the "Controller" in MVC whose responsibility is to handle the user input and manage the "Model" respectively. Controller component has different controllers or manager subcomponents including "FileManagement", "InputManagement", "GameManagement" and "MapManagement". In controller component, "MapManagement" component receives data from other manager subcomponents. Besides, "MapManagement" component has association with "Map" component that "MapManagement" modifies "Map" as MVC requires. Other management components accomplish different tasks such as reading and modifying files and listening to inputs. On the other hand, "GameManagement" component receives data from "MapManagement" component and starts the game if "Menu" from "User Interface" wants to do so.

3.3. Game Entities Subsystem

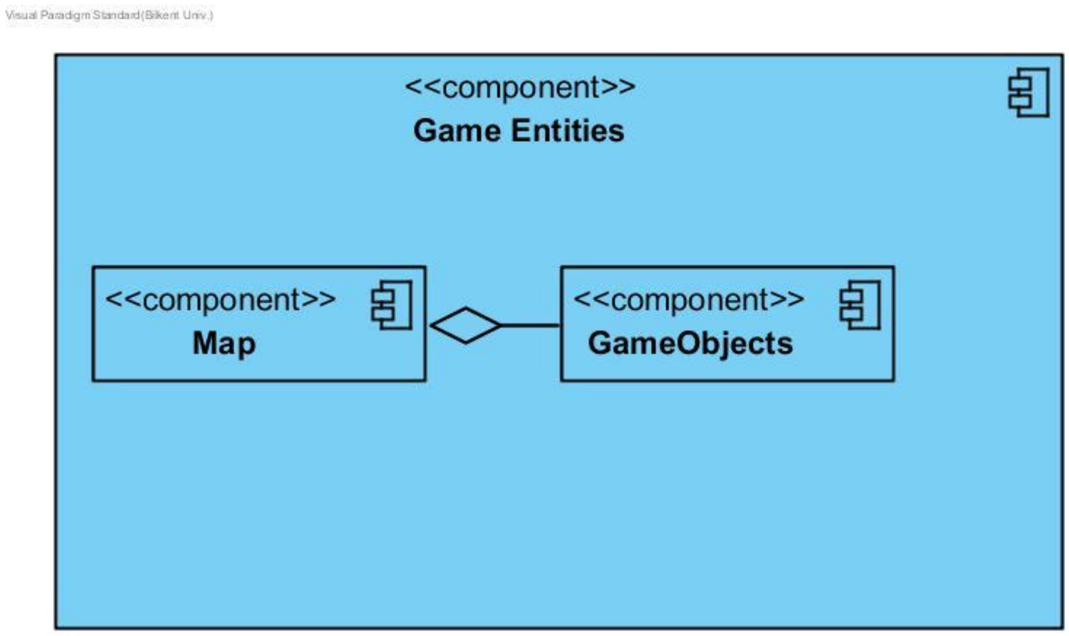


Figure 4: Game Entities Subsystem

Game Entities: At the very bottom is the "Game Entities" module as the "Model". The game objects related with neither "View" nor "Controller" are represented in this component. This module notifies the "User Interface" so that the "View" remains synchronized to actual data. In model component, we have Map which holds GameObjects and gets modified by MapManager. GameEntities can only be modified by Management Subsystem. Map component consists of all the game objects in the game.

4. Low-level design

4.1. Object design trade-offs

As we have already discussed our trade-offs in "Design Goals" section, in this part we will discuss the solutions to those trade-offs in "low-level" terms.

Memory vs. Time

Although memory was not specified explicitly in "Design Goals" section, memory consumption conflicts with speed and, thus, with response time.

However, as this course requires us to use as many different classes as possible (to see their interactions better), we did not care much about the memory consumption. Instead, we aimed at as many classes as possible. In other words, of course, knowing that this is a relatively simple project after all and our system will not use much of a space, still, we did not try to minimize the number of classes. Therefore, to achieve the required diversity of objects, we have come up with different children classes even when each can be refactored out to a general single class using an enumerated type. In other words, we needed to sacrifice memory by generating more classes than we normally should create.

Response Time vs. Modifiability

As mentioned in "Design Goals" section, response time is a greater concern of our project (rather than modifiability). Therefore, we tried to provide direct references to objects which need faster access during object design. This helps to prevent unnecessary calls -which otherwise make the function implementations more independent of one another- and provide a faster overall response for the core classes of our project.

4.2. Final object design

We have splitted the final class diagram according to our subcomponent decomposition. The following three are regarding the "Model" component's classes (first one is the general view, the following two are zoomed left and right sections):

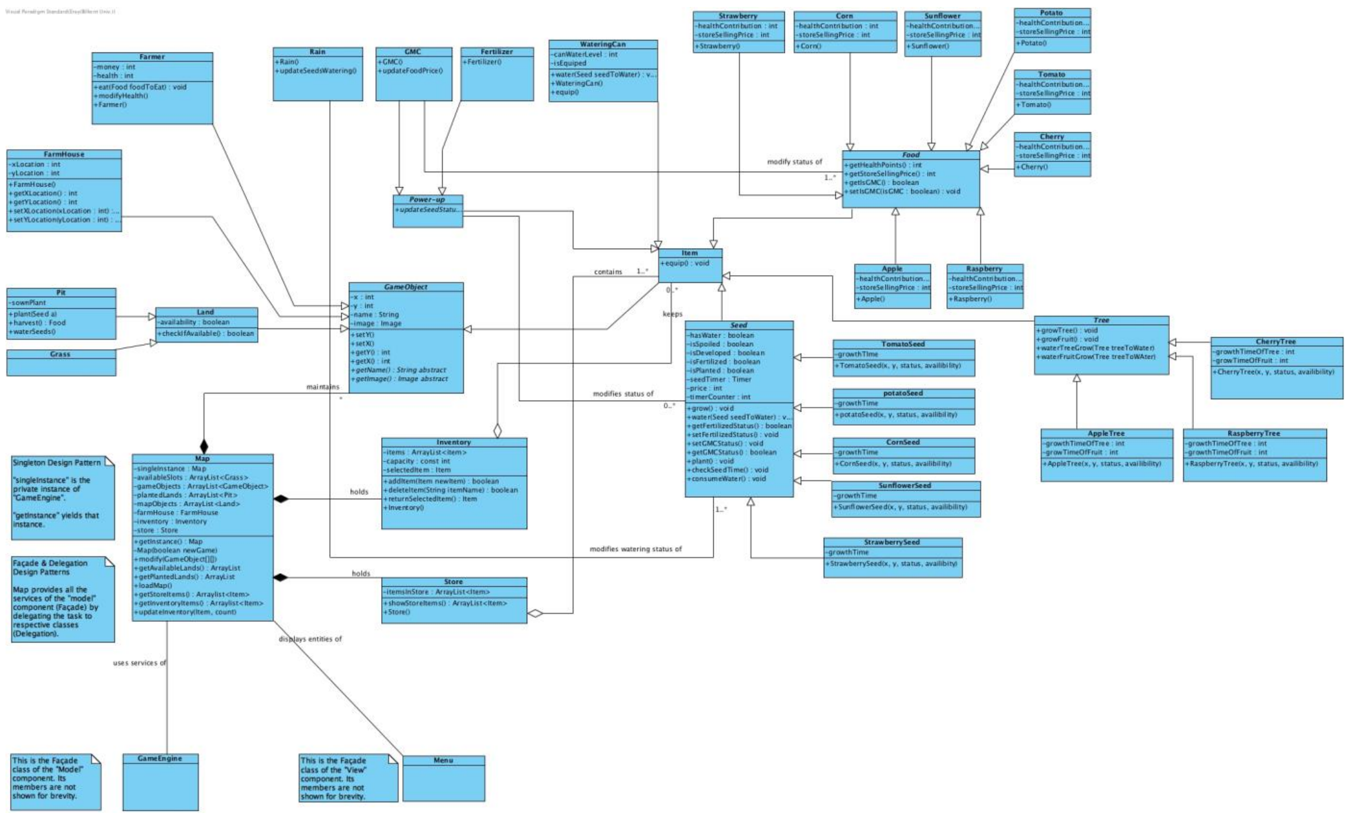

Farmer
FarmHouse
Pit
Grass
Land
Rain
GMC
Fertilizer
WateringCan
Power-up
GameObject
Item
Food
Strawberry
Corn
Sunflower
Potato
Tomato
Cherry
Apple
Raspberry
Seed
TomatoSeed
potatoSeed
CornSeed
SunflowerSeed
StrawberrySeed
Tree
CherryTree
AppleTree
RaspberryTree
Inventory
Store
Map
GameEngine
Menu
modify status of
contains
keeps
modifies status of
modifies watering status of
maintains
holds
holds
displays entities of
uses services of
Singleton Design Pattern
"singleInstance" is the private instance of "GameEngine".
"getInstance" yields that instance.
Façade & Delegation Design Patterns
Map provides all the services of the "model" component (Façade) by delegating the task to respective classes (Delegation).
This is the Façade class of the "Model" component. Its members are not shown for brevity.
This is the Façade class of the "View" component. Its members are not shown for brevity.

Figure 5.1: Game Entities Subsystem's Classes

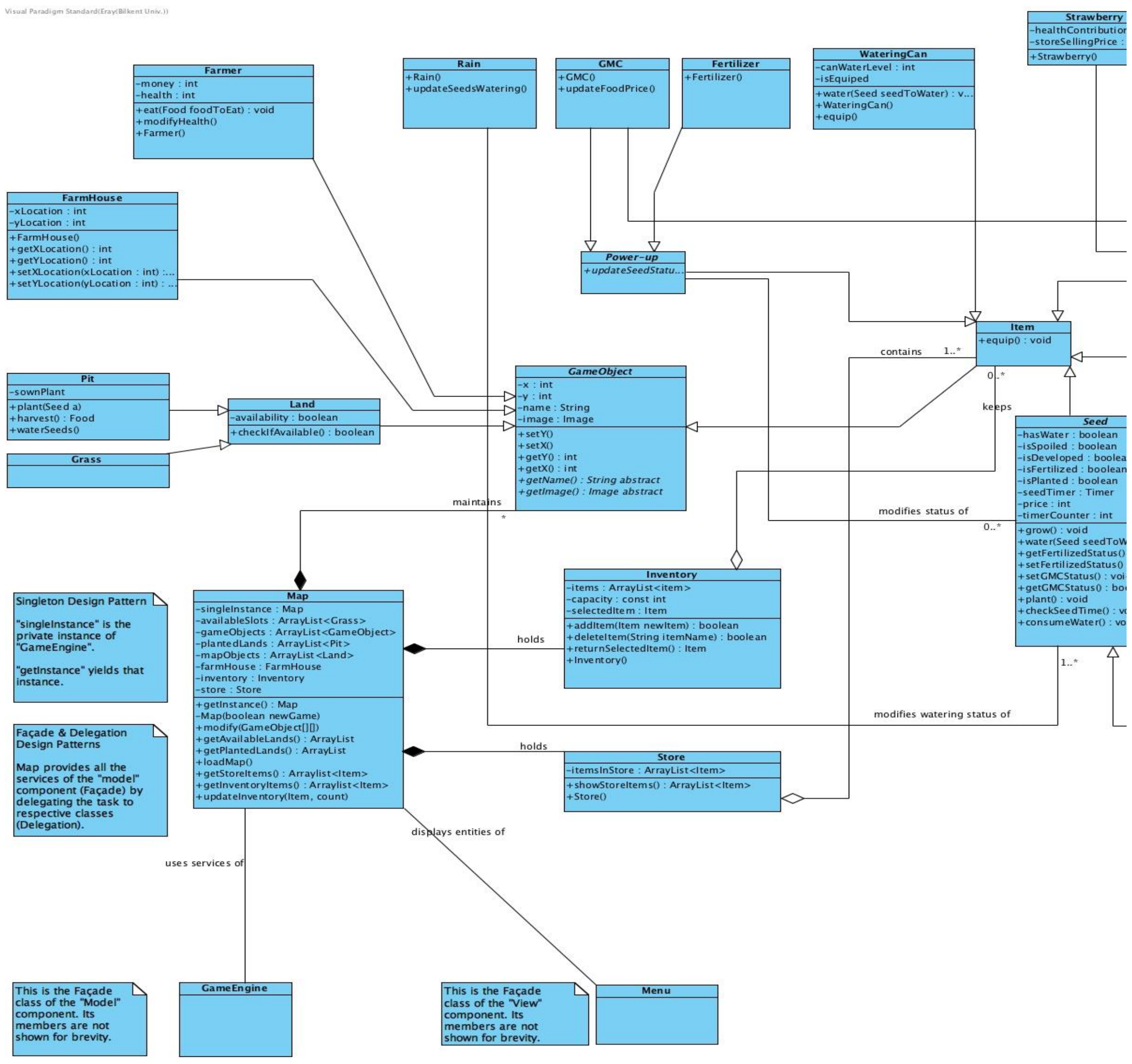


Figure 5.2: Game Entities Subsystem’s Classes (Left Part)

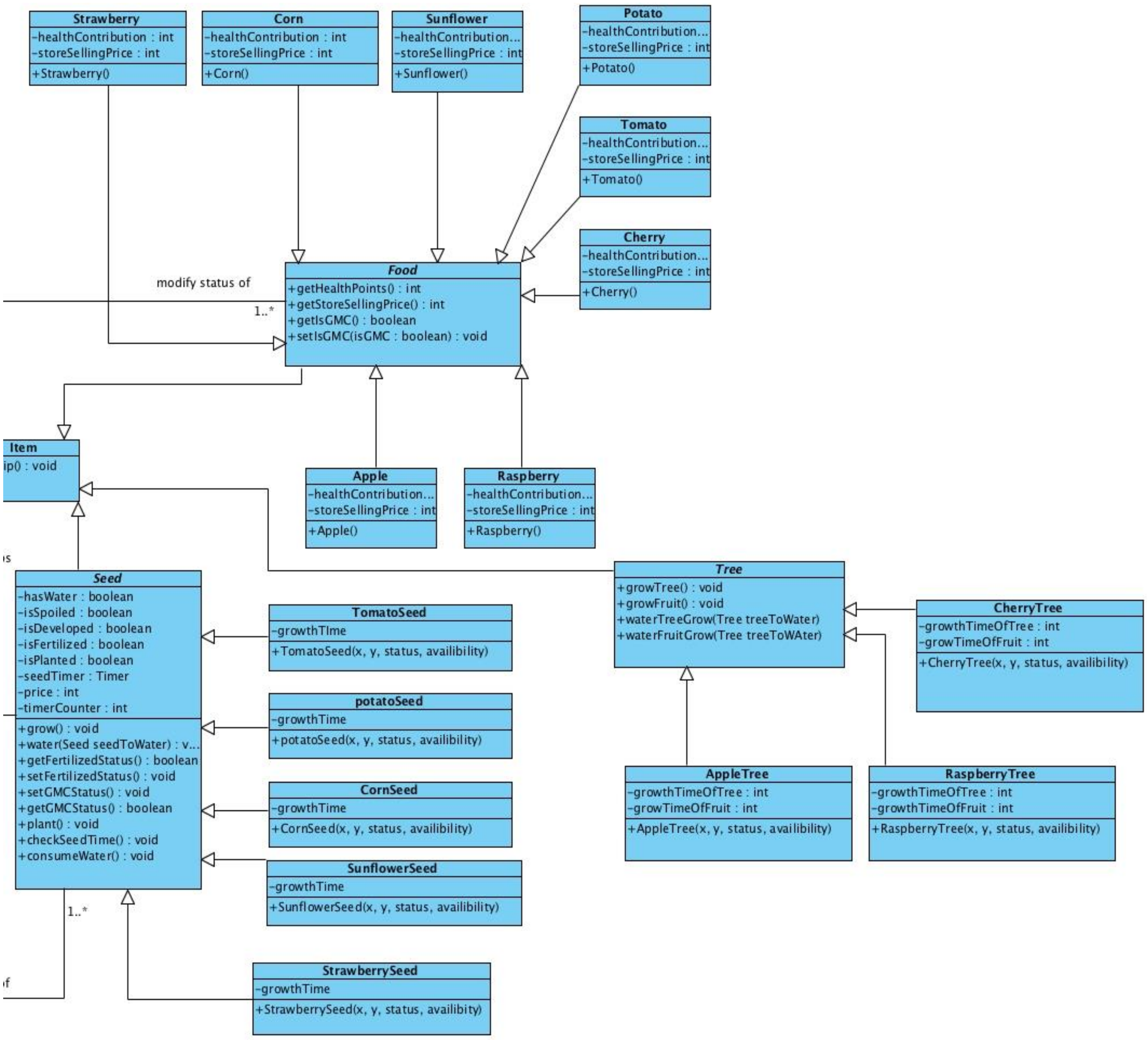
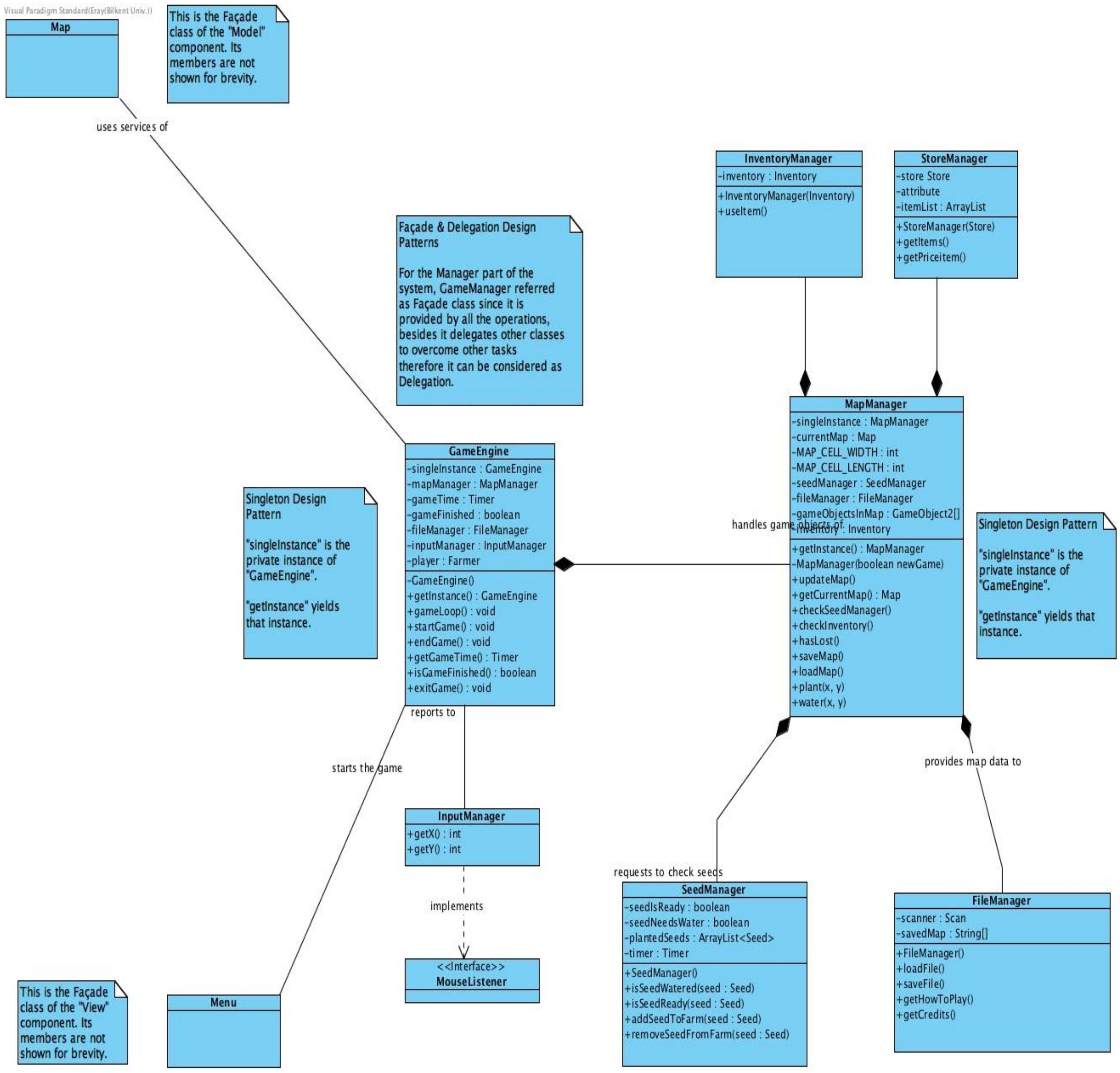


Figure 5.3: Game Entities Subsystem’s Classes (Right Part)

The class diagram of the “Controller” component follows:



This is the Façade class of the "Model" component. Its members are not shown for brevity.

Façade & Delegation Design Patterns

For the Manager part of the system, GameManager referred as Façade class since it is provided by all the operations, besides it delegates other classes to overcome other tasks therefore it can be considered as Delegation.

Singleton Design Pattern

"singleInstance" is the private instance of "GameEngine".

"getInstance" yields that instance.

Singleton Design Pattern

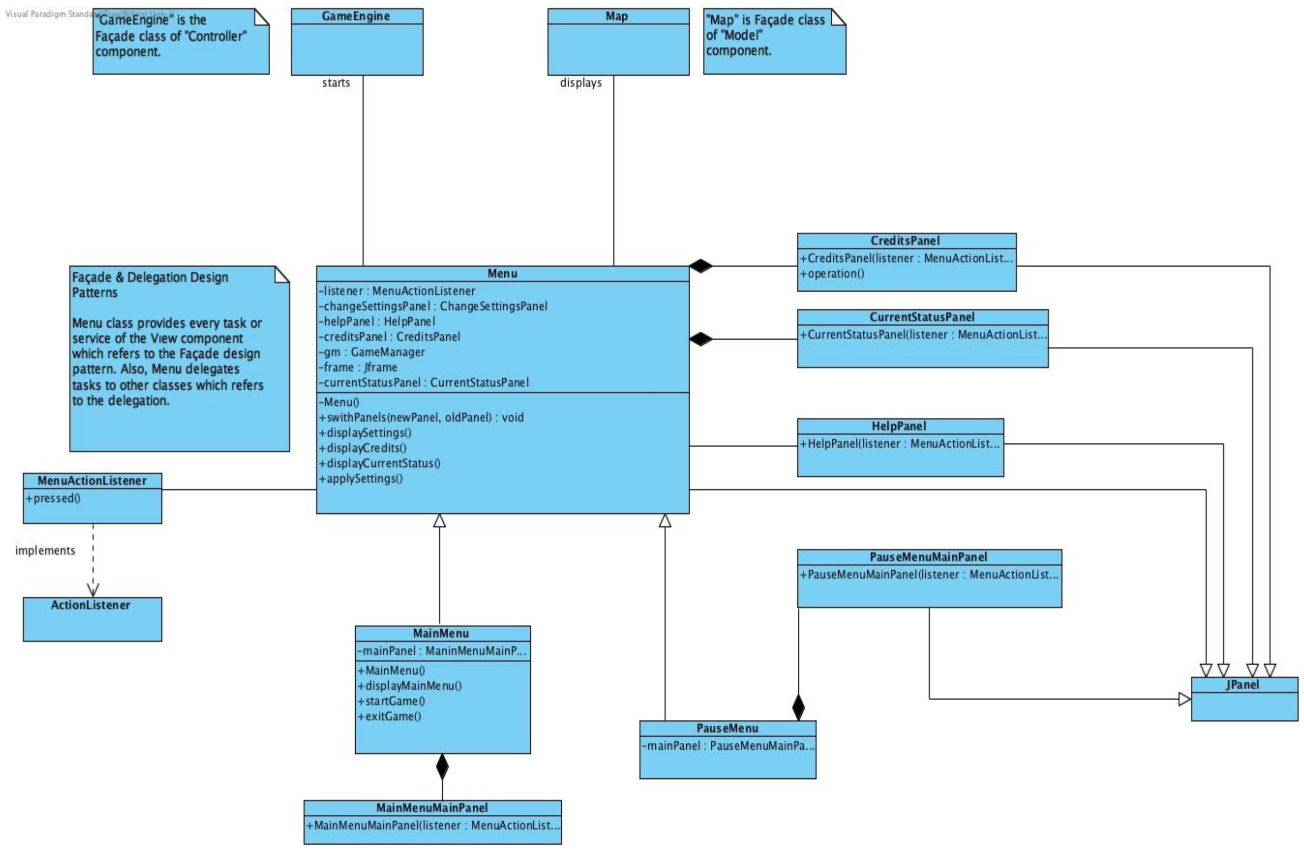
"singleInstance" is the private instance of "GameEngine".

"getInstance" yields that instance.

This is the Façade class of the "View" component. Its members are not shown for brevity.

Figure 6: Game Management Subsystem's Classes

Finally, the class diagram of the "View" component is given below:



Visual Paradigm Standard
"GameEngine" is the Façade class of "Controller" component.
GameEngine
starts
Map
displays
"Map" is Façade class of "Model" component.
Façade & Delegation Design Patterns
Menu class provides every task or service of the View component which refers to the Façade design pattern. Also, Menu delegates tasks to other classes which refers to the delegation.
Menu
-listener : MenuActionListener
-changeSettingsPanel : ChangeSettingsPanel
-helpPanel : HelpPanel
-creditsPanel : CreditsPanel
-gm : GameManager
-frame : Jframe
-currentStatusPanel : CurrentStatusPanel
-Menu()
+swithPanels(newPanel, oldPanel) : void
+displaySettings()
+displayCredits()
+displayCurrentStatus()
+applySettings()
CreditsPanel
+CreditsPanel(listener : MenuActionList...
+operation()
CurrentStatusPanel
+CurrentStatusPanel(listener : MenuActionList...
HelpPanel
+HelpPanel(listener : MenuActionList...
MenuActionListener
+pressed()
implements
ActionListener
PauseMenuMainPanel
+PauseMenuMainPanel(listener : MenuActionList...
MainMenu
-mainPanel : ManinMenuMainP...
+MainMenu()
+displayMainMenu()
+startGame()
+exitGame()
PauseMenu
-mainPanel : PauseMenuMainPa...
JPanel
MainMenuMainPanel
+MainMenuMainPanel(listener : MenuActionList...

Figure 7: User Interface Subsystem's Classes

Design Patterns

Singleton Design Pattern

We have decided to use Singleton Design Pattern in order to make sure core classes can be instantiated only once. We will have only one instance of "GameEngine" for checking the game and one "MapManager" for modifying the map (therefore, single "Map" instance, too). This will prevent us from any possible errors related with multiple instances and maybe more problematic cases. Yet, the issue with Singleton Design Pattern is that it can increase the coupling effect.

Façade Design Pattern

We have decided to use Façade Design Pattern in order to reduce coupling and to accomplish easier interaction between subsystems. Some classes in our code provide interfaces to other subsystems. For example, direct changes from “UserInterface” to “Management” must happen through “GameEngine” as its the Façade class there. If the player wants to start the game, then that request comes to “GameEngine” which starts other managers such as “MapManager”, “FileManager” and so on.

Delegation Design Pattern

Delegation Pattern emerges automatically as part of the Façade Design Pattern. Specifically, since Façade Design Pattern requires tasks to be handled by their respective classes, the Façade class there simply “delegates” the tasks to their actual owners. Thus, Delegation Design Pattern can be seen where Façade Design Pattern exists, which is already described above.

4.3. Packages

The packages are categorized into two as “Java Packages” and “directories”:

4.4.1. Java Packages

4.4.1.1. javax.swing

This package contains all the GUI related classes (JLabel,JFrame,JDialog, etc…).

4.4.1.2. java.util

This package contains array list for holding GameObjects.

4.4.1.3. java.awt

This package contains graphics objects for drawing and action listener classes for inputs.

4.4.1.4. java.io

This package contains classes for file reading operations and file exception handling.

4.4.2. Directories

Having composed the system into three main subsystems (according to MVC), there will be three respective folders as "model", "view" and "controller" (therefore, there is no additional package diagram).

4.4.2.1. Model directory

This directory includes all Game Object classes.

4.4.2.2. View directory

This directory includes all View classes (GamePanel,DrawPanel,Menus,etc..)

4.4.2.3. Controller directory

This directory includes management classes. (MapManager,FileManager,GameEngine)

4.4.2.4. Images directory

This directory includes all images(2d assets,background)

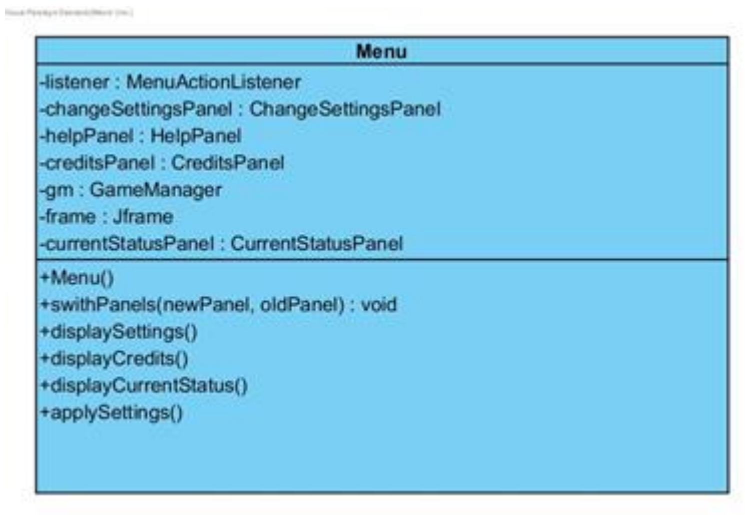
4.4.2.4. Data directory

This directory includes text files for save and load.

4.4. Class Interfaces

Detailed information about the classes (their interfaces) is given below:

Menu Class


Menu
-listener : MenuActionListener
-changeSettingsPanel : ChangeSettingsPanel
-helpPanel : HelpPanel
-creditsPanel : CreditsPanel
-gm : GameManager
-frame : Jframe
-currentStatusPanel : CurrentStatusPanel
+Menu()
+swithPanels(newPanel, oldPanel) : void
+displaySettings()
+displayCredits()
+displayCurrentStatus()
+applySettings()

Menu class is the class which has different classes that should be displayed inside of the menu. Attributes of this class is explained as following :

Attributes

Class Name : Menu	**Explanation of attributes**
listener : MauseActionListener	To listen related user action on Menu
changeSettingsPanel : ChangeSettingsPanel	ChangeSettingsPanel will provide the settings of the game
helpPanel: HelpPanel	Include main instructions of the game
creditsPanel : CreditsPanel	Credits interface will also be available

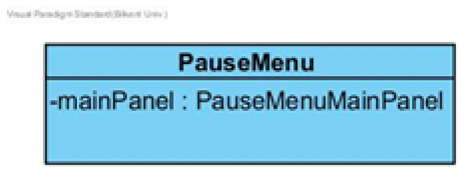
gameManager : GameManager	Game Manager will be controlling the game
frame : JFrame	Provides menu frame.
currentStatusPanel:CurrentStatusPanel	Show the current status of game.

These attributes will be used inside of the functions Menu class. These functions are explained as following :

Functions

Class Name : Menu	**Explanation of functions**
Menu():	This is the menu providing function.
switchPanels(newPanel, oldPanel):	If pressed to one button on the menu then this function will take the input and change the panel from the old panel.
displaySettings():	This will be used when settings button is pressed
displayCurrentStatus():	This function will be used when the settings button is pressed. This will navigate to the CurrentStatusPanel.
applySettings():	This will send the acitons that are made by user on Settings page to GameManager to change the current settings.

Pause Menu

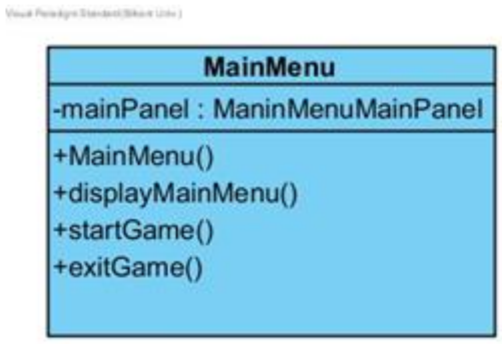

Visual Paradigm Standard(Bilkent Univ.)
PauseMenu
-mainPanel : PauseMenuMainPanel

This is the menu that will be able to be accessed from the game page. This Pause Menu will have following functions. This function will connect to the main menu so it's actions will be done.

Functions

Class Name : PauseMenu	**Explanation of functions**
mainPanel:PauseMenuMainPanel	This will navigate to mainPanel and will let user do main menu actions.

Main Menu


Visual Paradigm Standard(Bilkent Univ.)
MainMenu
-mainPanel : ManinMenuMainPanel
+MainMenu()
+displayMainMenu()
+startGame()
+exitGame()

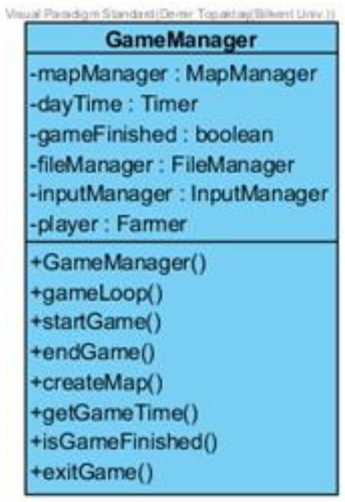
Main menu is the menu that will enable the user to interact with the game with selecting new game or reloading the last saved one, saving the current status of game, exiting game. Main Menu's functions are explained bellow :

Functions

Class Name : MainMenu	**Explanation of functions**
mainMenu():	Returns the current frame

displayMainMenu():	Sets a new frame.
startGame():	Refreshes the frame after setting one new frame.
exitGame():	Exits the game.
saveGame():	Saves the current status of game.

Game Manager Class



This class is to control the state of game and provides information about game state to other classes. Attributes are as following :

Attributes

Class Name : Game Manager	**Explanation of attributes**
private MapManager mapManager:	It is the instance of MapManager. It manages Map and provides the communication between GameManager and Map.
private Timer gameTime	This attribute keeps track of game time and it is initiliazed when the game starts.
private boolean gameFinished	gameFinished attribute keeps track of whether game is finished or not.
private FileManager fileManager:	This instance provides files to be saved or loaded to game manager
private InputManager inputManager	This instance controls the user mouse actions.
private Farmer player	It is the instance of the Farmer object and represents the player.

Functions

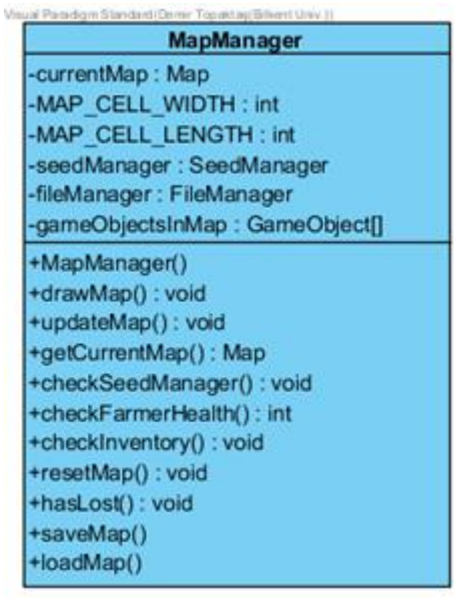
Class Name : Game Manager	Explanation of functions
private void startGame():	This method starts the game.
private void endGame():	This method ends the game.
private void exitGame():	This method exits the game.

private boolean isGameFinished():	This method returns the game finished attribute
private Timer getGameTime():	This method returns the game time timer.
p**rivate void saveGame():**	This method saves the game to a file

Constructor

public GameManager(): Default constructor for Game Manager. It initiliazes the GameManager object.

MapManager Class



MapManager class controls, updates, and modifies the map accordingly to relations with other classes.

Attributes

Class Name : MapManager Class	**Explanation of attributes**
private Map currentMap	This is the instance of Map object
private int MAP_CELL_WIDTH & MAP_CELL_LENGTH	These attributes are for dividing map equally.
private SeedManager seedManager	This is the instance of SeedManager. It is responsible for checking “planted” seeds status (watered or not) in the map.
private FileManager fileManager	It is the instance of FileManager and it provides saved files to Map Manager

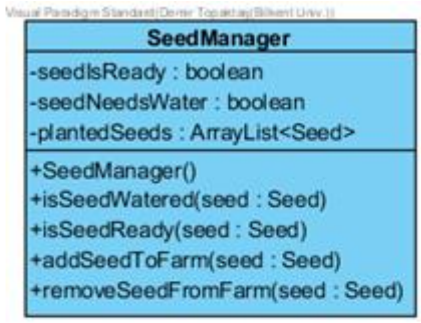
Constructor

public MapManager(): Default constructor for MapManager.

Functions

Class Name : MapManager Class	**Explanation of function**
private void drawMap():	This method draws the initiliazed map.
private void updateMap()	This method updates the map
private void loadMap():	This method loads the map from a file using FileManager
private void saveMap():	This method saves the map to a file using FileManager.
private Map getCurrentMap():	This method returns the current map object
private void checkSeedManager():	This method requests SeedManager to check seeds in the farmland
private boolean hasLost():	This method returns whether the game is lost or not.
private void resetMap():	This method resets the map.
private void checkFarmerStatus():	This method checks farmer status such as farmer's health and money

SeedManager Class


Visual Paradigm Standard(Deniz Topaktaş(Bilkent Univ.))
SeedManager
-seedIsReady : boolean
-seedNeedsWater : boolean
-plantedSeeds : ArrayList<Seed>
+SeedManager()
+isSeedWatered(seed : Seed)
+isSeedReady(seed : Seed)
+addSeedToFarm(seed : Seed)
+removeSeedFromFarm(seed : Seed)

SeedManager is responsible for checking planted seeds status in the map. Seeds can be planted in the map at different times and have different growth status. Some seeds might need water or ready to be harvested. Seed manager checks these attributes.

Attributes

Class Name : SeedManagerClass	**Explanation of attributes**
private ArrayList<Seed> plantedSeeds:	This array list keeps the seeds that are planted to the map.
private boolean seedIsReady:	This attribute checks whether seed is ready to be harvested or not.
private boolean seedNeedsWater:	This attribute checks whether seed needs water or not.

Constructor

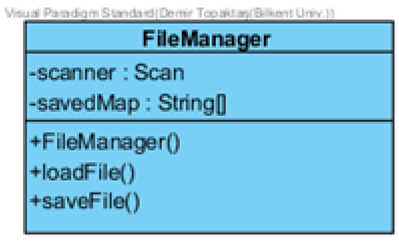
public SeedManager(): Default constructor for seed manager.

Functions

ClassName :SeedManager Class	**Explanation of functions**
private boolean isSeedWatered(Seed seed):	This is the method for checking specific seed is watered or not.
private boolean isSeedReady(Seed seed):	This is the method for checking specific seed is ready for

	harvesting.
private void addSeedToFarm(Seed seed):	This method adds seed to array list of planted seeds.
private void removeSeedFromFarm(Seed seed):	This method removes seed from array list.

File Manager Class



FileManager will be responsible for saving or loading the game. It will save game to file or it will load game from file.

Attributes

ClassName :FileManager Class	Explanation of attributes
private String[][] map:	This attribute holds the map in 2d string array
private Scanner scan:	This attribute is for scanning.

Constructor

public FileManager(): Default constructor for Game Manager. It initiliazes the GameManager object.

Functions

ClassName :FileManager Class	**Explanation of attributes**
private loadFile():	This methods reads from the saved map from file and loads.
private saveFile():	This method saves the map to the file.

InputManager Class

This class will be responsible for user inputs. It will implement MouseListener to detect mouse clicks

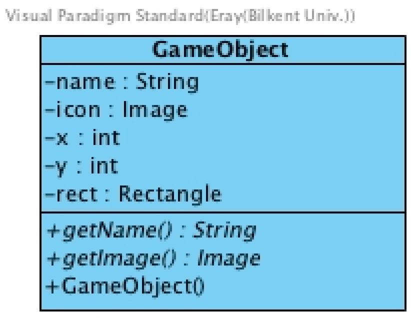
Constructor

public InputManager(): Default constructor for Game Manager. It initiliazes the GameManager object.

Methods

public void mouseClicked(): This method initiates when mouse clicks**.**

GameObject Class



This is an abstract class to generalize the concept of game objects.

Attributes

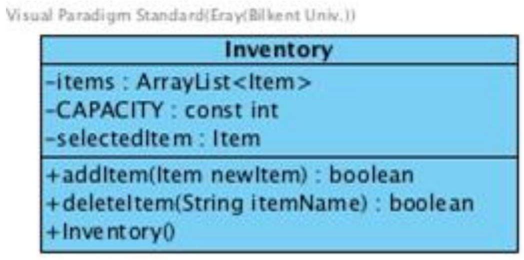
ClassName : GameObjectClass	**Explanation of attributes**

private int x;	is the x-coordinate of a *GameObject* (on the screen).
private int y;	is the y-coordinate of a *GameObject*.
private Rectangle rect	is used to define the boundary areas of the objects (for the game logic).
private String name	is the *name* of the *GameObject*. Specifically, it is the type (such as "*CornSeed*").
public GameObject(int x, int y)	constructs a *GameObject*, assigns *x* and *y* values to the ones provided. It also creates a *Rectangle* object (which is assigned to *rect*).

Functions

ClassName : GameObject Class	**Explanation of functions**
public abstract String getName();	is the abstract method to return the *name*, to be implemented accordingly in child classes
public abstract Image getImage();	is, similarly, another abstract method that returns the *icon* of a *GameObject* instance. This will, also, be implemented differently in child classes.

Inventory Class


Visual Paradigm Standard(Eray(Bilkent Univ.))
Inventory
-items : ArrayList<Item>
-CAPACITY : const int
-selectedItem : Item
+addItem(Item newItem) : boolean
+deleteItem(String itemName) : boolean
+Inventory()

This class represents the inventory where the purchased seeds and collected food are stored. *Inventory* class is one of the children of *GameObject*.

Attributes

ClassName : InventoryClass	**Explanation of attributes**
private ArrayList<Item> items;	is an array list to keep *Item* objects (will be used to store *Seed* and / or *Food* instances).
private const int CAPACITY;	defines the maximum number of *Item* instances that can be stored in the inventory.

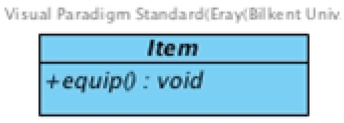
private Item selectedItem;	is the *Item* instance that is currently chosen / equipped.

Functions

ClassName : InventoryClass	**Explanation of functions**
public boolean addItem(Item newItem);	receives an *Item* instance and adds it to the array list. If the *CAPACITY* is already full, returns false to indicate this unsuccessful attempt (returns true for the successful case).
public boolean deleteItem(String itemName);	given the *name* of an *Item* object as a string, for instance let "corn" be received through the parameter, decreases the *count* attribute of that *Item* by 1. More specifically, if the *Item* instance in the list whose type is received as a string has a quantity (*count*) more than 1, then *count* gets decremented after calling this function. Otherwise, when the *Item* instance has *count* equal to 1, the instance of that *Item* is deleted from the array list. Returns true when an *Item* instance gets deleted or its *count* is decremented (returns false when

such an *Item* does not exist).

Item Class

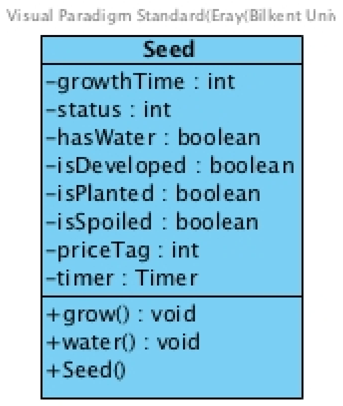


Being another child class of *GameObject*, this abstract class is the parent of *Seed*, *Food* and *WateringCan* classes.

Functions

ClassName : ItemClass	**Explanation of functions**
public abstract void equip();	will be used to equip an *Item* instance (*selectedItem* in the *Inventory* will be adjusted).

Seed Class (and its children)



The *Seed* class is the superclass of its variations (*StrawberrySeed*, *CornSeed*, *SunflowerSeed, TomatoSeed* and *PotatoSeed*) and is a child of the *Item* class. Since there are many shared properties among all kinds of seeds, this generalization of a *Seed* superclass proves useful.

Attributes

ClassName : SeedClass	**Explanation of attributes**
private int growthTime;	is the time required for a Seed instance to grow fully.
private int status;	demonstrates the status of a *Seed* in terms of growth.
private boolean hasWater	keeps whether a *Seed* instance has been watered.
private boolean isDeveloped	indicates if a *Seed* object is developed.
private boolean isPlanted	is used to understand if a *Seed* object is planted or not.

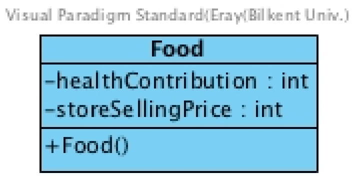
private boolean isSpoiled	is true when a *Seed* gets spoiled, false otherwise.
private boolean isFertilized;	keeps whether this *Seed* object is fertilized.
private int priceTag;	is the amount of money required to buy a *Seed* from the *Store*.
private Timer timer	is kept to measure the time after a *Seed* is being planted.

Note that all *Seed* types (i.e. the child classes) have the same attributes as those above. However, *growthTime* and *priceTag* will be overridden in each subclass.

Functions

ClassName : SeedClass	**Explanation of functions**
public void grow();	will be called to let a *Seed* instance grow, adjusting the status.
public void water();	helps to water a *Seed* instance.

Food Class (and its children)



Visual Paradigm Standard(Eray(Bilkent Univ.)
Food
-healthContribution : int
-storeSellingPrice : int
+Food()

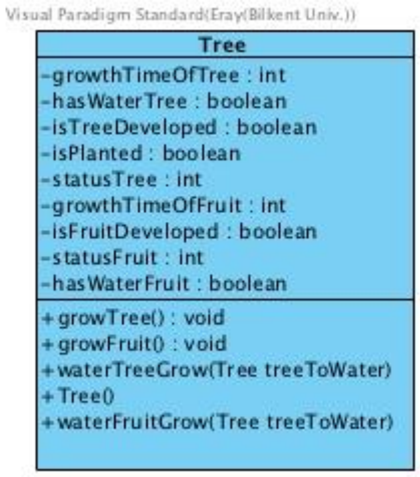
Similar to the *Seed* class, *Food* is a child of *Item*. Moreover, its kinds (*Strawberry*, *Corn*, *Sunflower, Potato, Tomato, Cherry, Raspberry* and *Apple*) are represented as the children of this class.

Attributes

Class Name : Food Class	**Explanation of attributes**
private int healthContribution;	is how much a *Food* instance, when eaten, will add up to the *health* of the *Farmer*.
private int storeSellingPrice;	is, similarly, how much *money* will be received by the *Farmer* when a *Food* is sold.

Again, the child classes (classes representing different kinds of *Food*) have the same attributes. However, each of them will be overriding both the *healthContribution* and *storeSellingPrice* (not shown for easier understanding).

Tree Class (and its children)



Tree class is the superclass of *AppleTree*, *RaspberryTree* and *CherryTree*.

Attribute

ClassName: TreeClass	**Explanation of attributes**
private int growthTimeOfTree;	is the growth time of a tree.
private boolean hasWaterTree;	keeps if a *Tree* instance is watered.
private boolean isTreeDeveloped;	keeps whether a *Tree* instance is developed.
private boolean isPlanted;	is used to understand if a *Tree* object is planted somewhere.
private int statusTree;	defines the status of the tree in terms of growth.
private int growthTimeOfFruit	keeps the growth time of the fruit which is generated from a tree.
private boolean isFruitDeveloped;	holds whether a *Tree* instance has formed any fruits (*Food*).

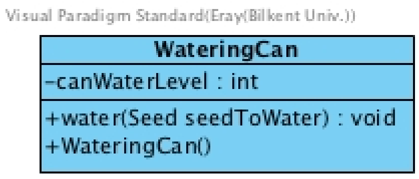
private int statusFruit;	keeps the status of the fruit on the tree.
private boolean hasWaterFruit;	tracks the water condition of the fruit.

Functions

ClassName : TreeClass	**Explanation of functions**
public void growTree();	is called to let a *Tree* instance grow.
public void growFruit();	depending on the tree's development, this function will help produce a fruit.
public void waterTreeGrow(Tree treeToWater);	is used to water a tree which has not produced any fruits yet.
public void waterFruitGrow(Tree treeToWater);	is used to water a tree that already has produced fruits.

The subclasses of the *Tree* class have exactly the same members as their parent. However, again, they will be overloading particular members depending on the type of the *Tree* object.

WateringCan Class


Visual Paradigm Standard(Eray(Bilkent Univ.))
WateringCan
-canWaterLevel : int
+water(Seed seedToWater) : void
+WateringCan()

As the name suggests, this is the class representing a watering can. Again, this class is a subclass of *Item*.

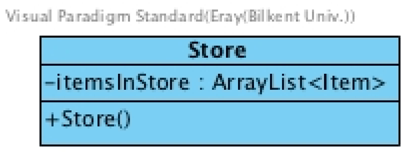
Attributes

ClassName : WateringCanClass	**Explanation of attributes**
private int canWaterLevel;	is the amount of the available water.

Function

ClassName : FarmerClass	**Explanation of function**
public void water(Seed seedToWater);	calls the *water()* function of the received *Seed* instance and adjusts the *canWaterLevel*.

Store Class

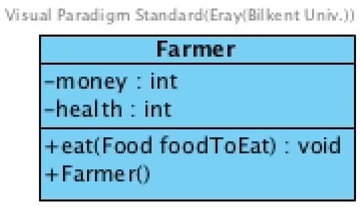

Visual Paradigm Standard(Eray(Bilkent Univ.))
Store
-itemsInStore : ArrayList<Item>
+Store()

This is another child of the *GameObject* class and it represents the store in the game.

Attribute

ClassName : StoreClass	**Explanation of attributes**
private ArrayList<Item> itemsInStore;	keeps the *Item* objects that are available in the store.

Farmer Class


Visual Paradigm Standard(Eray(Bilkent Univ.))
Farmer
-money : int
-health : int
+eat(Food foodToEat) : void
+Farmer()

This class represents the *Farmer*.

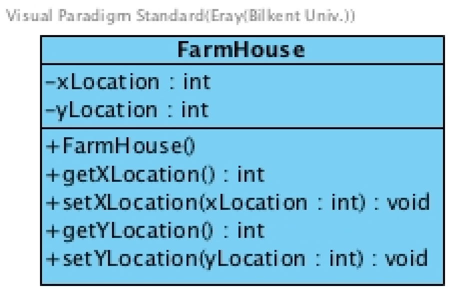
Attributes

ClassName : FarmerClass	**Explanation of attributes**
private int money;	**is the amount of *money* that the *Farmer* has.**
private int health	represents the *health* level of the *Farmer*.

Functions

ClassName:	Explanation of the functions
public void eat(Food foodToEat);	given a *Food* instance via the parameter, the Farmer's *health* gets incremented according to that *Food*'s *healthContribution*.

FarmHouse Class


Visual Paradigm Standard(Eray(Bilkent Univ.))
FarmHouse
-xLocation : int
-yLocation : int
+FarmHouse()
+getXLocation() : int
+setXLocation(xLocation : int) : void
+getYLocation() : int
+setYLocation(yLocation : int) : void

This is the class illustrating the *FarmHouse*. Being a child of *GameObject*, the inherited members are to be used.

Land Class

As a child of the *GameObject* class, this class is the parent of *Grass* and *Pit*. This abstract class does not have any members except those inherited from the *GameObject* class.

Grass Class

This class illustrates a type of Land that is not suitable for planting.

Pit Class

In contrast with *Grass*, this class demonstrates the kind of *Land* available for planting.

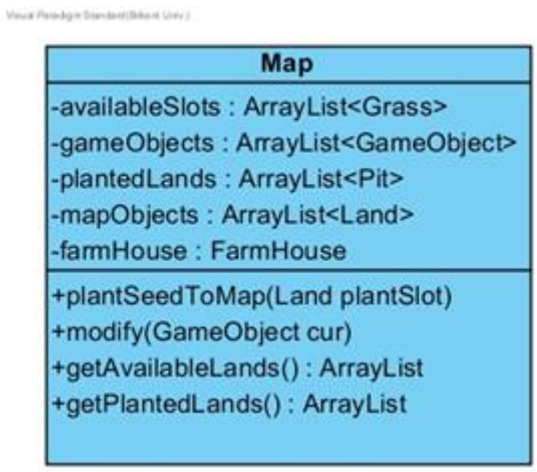
Attribute

private Seed sownPlant	keeps the *Seed* that is planted.

Functions

ClassName: Pit Class	**Explanation of functions**
public boolean plantSeed(Seed toPlant)	receives a *Seed* object to assign it to *sownPlant*. When planting is successful, true is returned. Otherwise, if the *Pit* is already planted for instance, this operation returns false.
public Food harvest()	when called on a planted *Pit* object, returns the *Food* of that *Seed*. Note that the *Seed* planted in that *Pit* object should have been already grown in order to be harvested.
public void waterSeeds()	is used to water the *Seed* instance on a *Pit* object.

Map Class


Visual Paradigm Standard(Bilkent Univ.)
Map
-availableSlots : ArrayList<Grass>
-gameObjects : ArrayList<GameObject>
-plantedLands : ArrayList<Pit>
-mapObjects : ArrayList<Land>
-farmHouse : FarmHouse
+plantSeedToMap(Land plantSlot)
+modify(GameObject cur)
+getAvailableLands() : ArrayList
+getPlantedLands() : ArrayList

This class is to keep objects in a grouped manner for better usage in *MapController*.

Attributes

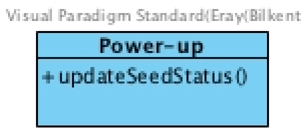
ClassName : MapClass	**Explanation of attributes**
private ArrayList< Grass> availableSlots	This ArrayList is to keep available *Grass* slots to change them into plantable *Pit* objects to plant *Seed* instances.
private ArrayList< GameObject> gameObjects	This list is to keep track of all objects by storing them.

private ArrayList< Pit> plantedLands	To keep track of where *Seed* objects have been planted.
private ArrayList< Land> mapObjects	To keep track of all Grass, *Pit*, in other words *Land* objects.
private FarmHouse farmHouse	This is an instance of the *FarmHouse* class which will be kept in the *Map* class.

Functions

Class Name : Map Class	**Explanation of functions**
public void plantSeedToMap(Land plantSlot)	This operation is to plant *Seed* to specific *Land* slots, and modify ArrayLists.
public void modify(GameObject cur)	This operation is to change selected *GameObject* to *GameObject cur* by the commands from game logic.
public ArrayList<Grass> getAvailableLands()	To return *availableSlots* to *MapController* class for using it in Game Logic. Especially, for separating planted lands from plantable lands.
public ArrayList<Pit> getPlantedLands()	This operation is to return *plantedLands* to *MapController* for a better understanding of the separation between plantable and planted lands like *getAvailableLands*().

Power-up Class

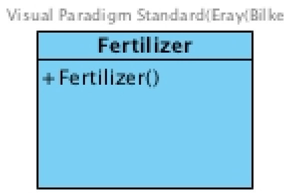


This is the parent of power-ups which are *Fertilizer* and *GMC* (genetically modified crop).

Functions

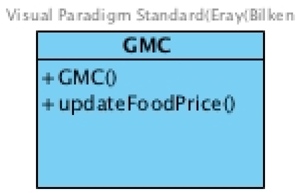
ClassName:Power-upClass	**Explanation of the functions**
public void updateSeedStatus()	to update the *Seed* objects' *status* after a power-up is applied.

Fertilizer Class



This class, as a child of *Power-up*, operates on *Seed* objects. After being purchased from the store and applied on a planted slot, then that specific farm slot becomes fertile (meaning that the seeds would grow faster on that slot).

GMC Class

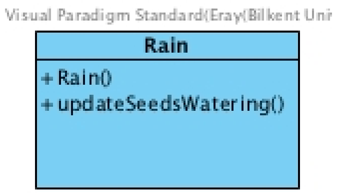


Standing for "Genetically Modified Crop," this is another child of the *Power-up* class. After it is bought from the store, it is applied on *Seed* instances.

Functions

ClassName : GMC Class	**Explanation of function**
public void updateFoodPrice()	decrements the *storeSellingPrice* of *Food* instances that are generated from genetically modified *Seed* instances (as the *Food* becomes less healthy, it worths less).

Rain Class



Being a child of the *GameObject* class, this class acts on *Seed* objects to change their water condition. More specifically, when this power-up is purchased once, the planted seeds may become watered at an unknown random time.

Functions

ClassName : Rain Class	**Explanation of functions**
public void updateSeedsWatering()	is called at a random time to water the planted seeds.

5. Improvement Summary

- We have changed class diagrams according to new design patterns (Singleton, Façade and Delegation)
- In the low level design, we have updated the class diagrams’ attributes and operations.
- In subsystem decomposition, we have explicitly explained subsystems and their relations.
- We have updated design goals and made them more coherent.