CS 319 - Object-Oriented Software Engineering
System Design Report

Iteration 1

Farmio

Group 2F

Nursena Kal

Demir Topaktaş

Ahmed Fuad

Eray Şahin

Context

1. Introduction

1.1 Purpose of the system

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.

1.2 Design Goals

Before the composing the system it is crucial to identify the design goals of the system in order to clarify the qualities that our system should focus on. In this respect many of our design goals inherit from non-functional requirements of our system that are provided in analysis stage. Crucial design goals of our system are described below.

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 satisfiying. Also it will be important to have great 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 : Since player is not ought to have knowledge about how the game is played, how to plant a seed, how to water and grow a seed. Also there will be a litttle instruction set in the begining 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:

Extendibility: In general, in the lifetime of game software, it is always important to add new components, features to the game in order to sustain the excitement and interest of the player. In this respect our design will be suitable to add new functionalities, entities (i.e. new brick types, new power-ups) easily to the existing system.

Portability: Portability is an important issue for a software, since it provides that the software can reach wide range of users. In this respect we are determined that the system will be implemented in Java, since its JVM provides platform independency, our system will satisfy the portability.

Modifiability: In our system it would be easy to modify the existing functionalities of the system. In order to achieve this we will minimize the coupling of the subsystems as much as possible, to avoid great impacts on system components by a desired change.

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 Ease of Learning vs. Functionality:

Since we designed this game as a simulation game, it is important to have a game which is easy to discover. However, in this game there are huge variety of objects such as different seeds, power ups, trees and tools that are used on these crops, it is a trade-off for us to make the game a little bit complicated. We choosed to make Farmio a little bit complicated but still easy to learn and observe.

Performance vs. Memory:

In our system, it is our main purpose to make the game as close to real life as possible. Briefly, performance of our system is our primary focus. For this purpose, we sacrificed the memory in order to gain the performance. We used a huge varity of objects. Therefore, our game has a lot to store. Since the performance of this event is important in our system, instead of storing one type of game entitiy, we store the important game objects of the game map in memory to access them fast when needed.

1.3 Definitions, acronyms, and abbreviations

Abbreviations:

MVC: [2] Model View Controller

JDK: [1] Java Development Kit

JVM: [1] Java Virtual Machine

1.4. References

[1] Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010, ISBN-10: 0136066836.

[2] FarmVille Game by Zynga : https://www.zynga.com/games/farmville

1.5. Overview

In this section Farmio game is briefly explained. The way it is designed and the considered actions are stated. Since this game is a simulation game, there is a huge variety of objects, it makes our game little bit complicated than we want but it is a tradeoff that we have already think is okay. The advantage of this game is that it is a simulation game, we are

observing the real world relations and implementing them into computer as a game. This makes our game easier to understand eventhough it has lots of different objects.

2. Software Architecture

2.1 Overview

This is the explanation part of inner side of the game.

2.1.1 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.

2.1.2. 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).

2.1.3. 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.

2.1.4. 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.

2.1.5. Seeds

The game represents strawberry, corn, sunflower tomato and potato 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.

2.1.6. Grown Crops

As already mentioned above, seeds produce grown crops when treated properly. These crops need to be harvested as soon as possible, otherwise they will rot, which is indicated by the change of the color of the soil.

2.1.7. 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. To harvest the grown crops, the player simply clicks on these new icons.

2.1.8. 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.

2.1.9. 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.

2.1.10 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.

2.1.11 Power-Ups

To add different style to Farmio, there are three different power-ups designed inside it. Rain, GMC and fertalizers. Rain will be randomly gained and it is used to change the watered status of seeds'. GMC and fertalizers will be available on the store. Their difference will be the selling prices and their effects on the seeds. GMC will be cheaper than fertalizers and lower the growing time but that will also lower the value of the gathered crops that will be sold. Fertalizers will cost more than GMC and they will lower the growing time of the seed and it will not lower the selling price.

2.2 Subsystem Decomposition

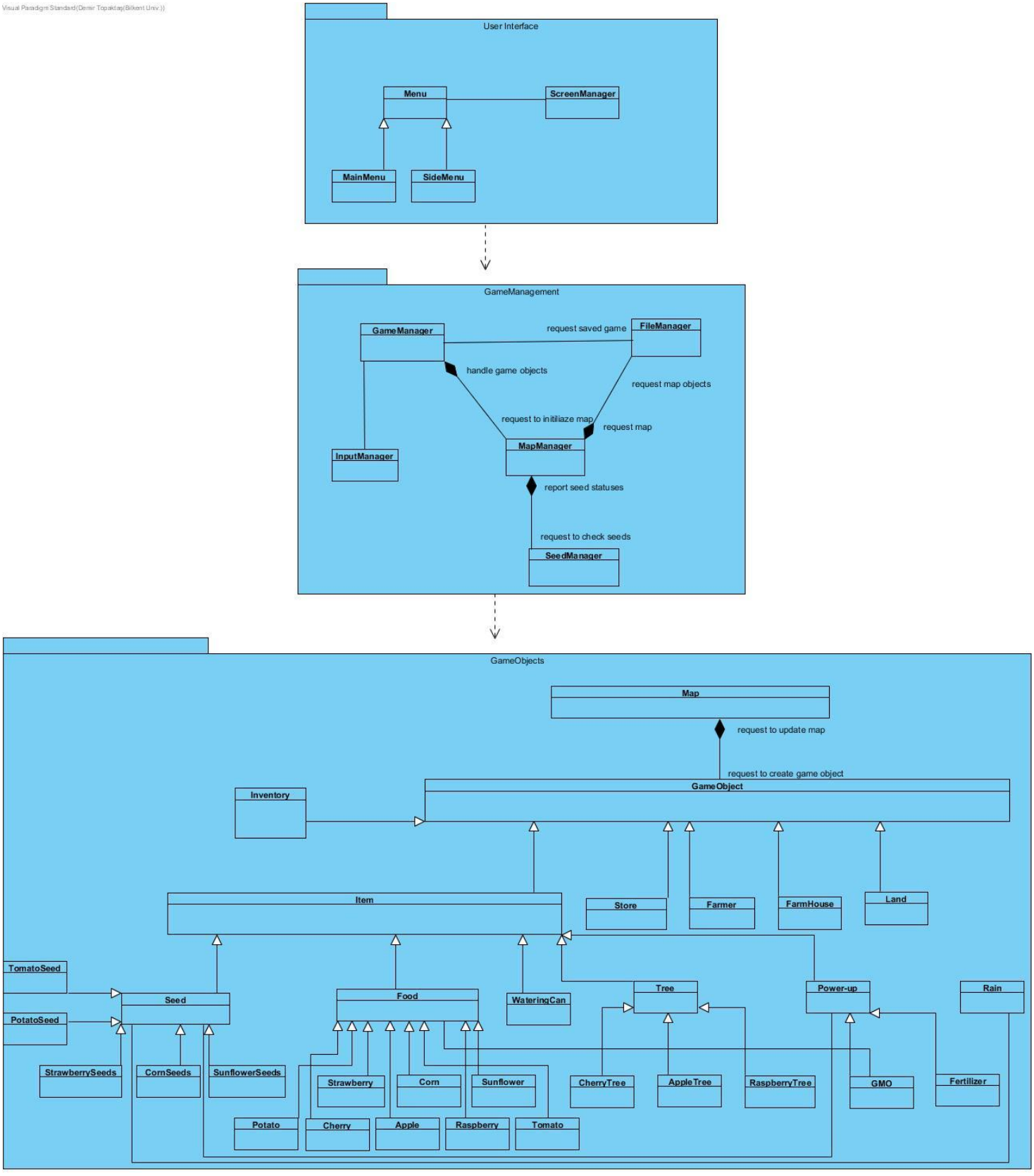


Figure 1 : Basic Subsystem Decomposition

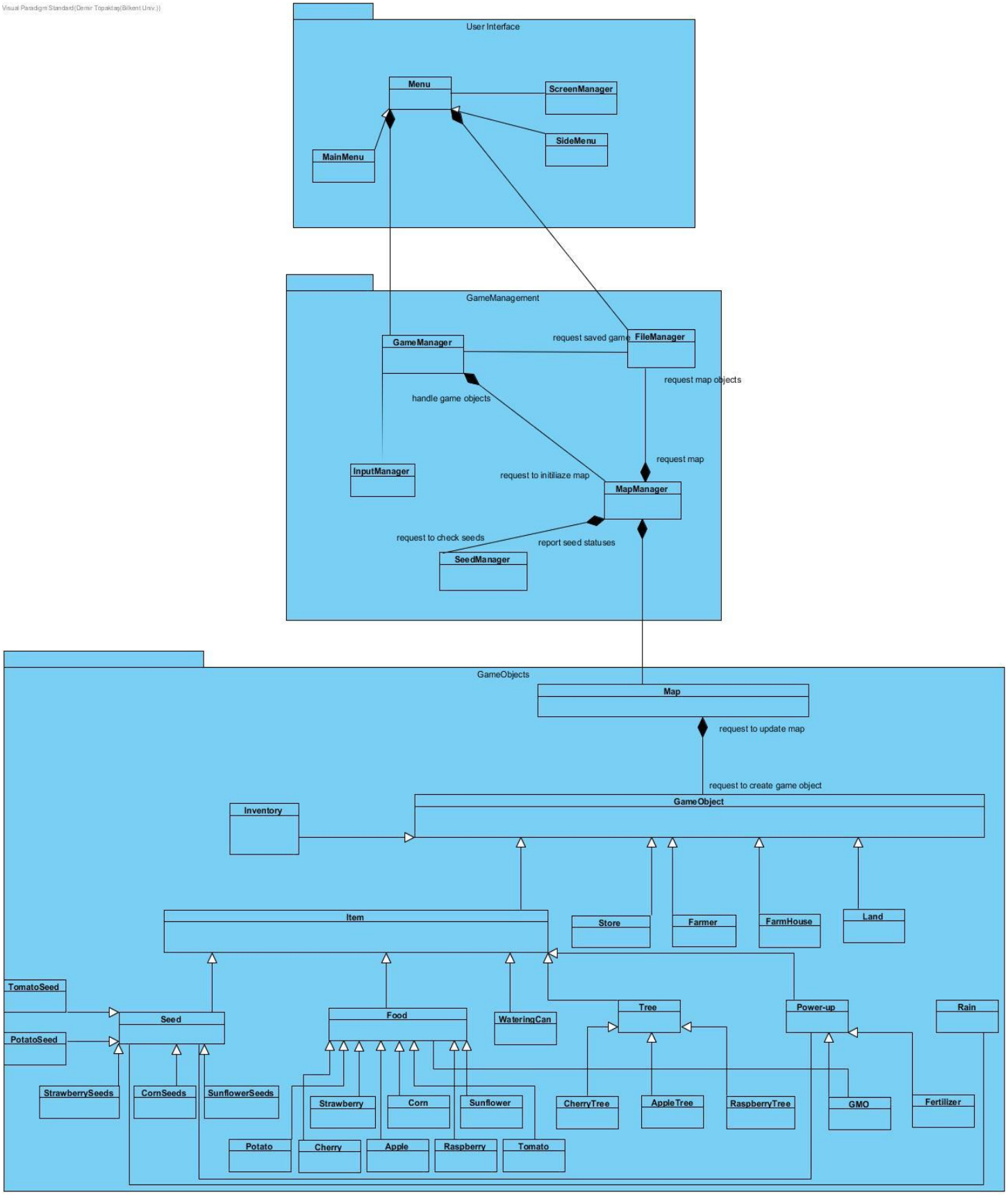


Figure 2: Basic Subsystem Decomposition Detailed

In this part, we will decompose our system into three parts; User Interface, Game Management and Game Objects. With this approach, we can reduce the complexity of the game and make the system eligible to changes. We will achieve high coherency and reduce coupling.

MVC(Model-View-Controller) architectural pattern will be chosen for this system. The system will be decomposed considering this pattern. User Interface subsystem will be our view and it will provide interface to the user. It will contain Menu, Settings etc. Our Game Management subsystem will be the controller for the system. It will control the game and it will maintain communication between Model and View. For example, map controller will control the map and game engine will be responsible for controlling the game. Game objects and map will be the Model part of the system.

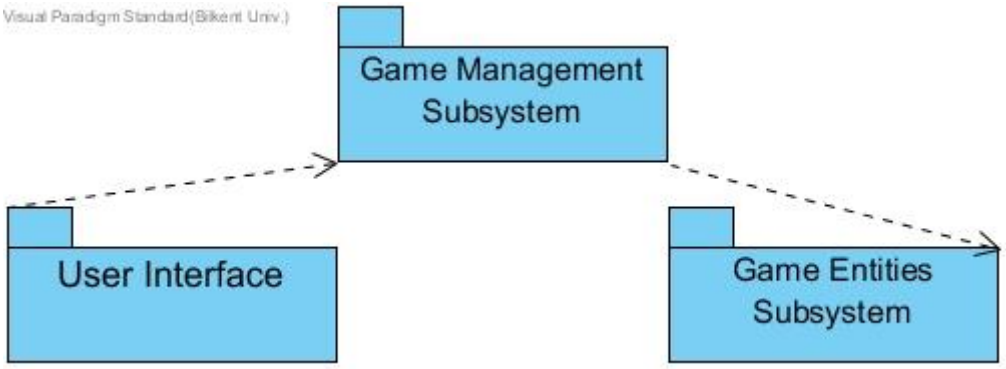
Subsystems will be separate from each other but they will work closely. User Interface will get the user input and transmit this information to the Game Management system. Then, Game Management will update Game Objects according to the information. Communication between User Interface and Game Objects will be happened through Game Management(Façade Design). This design principle will provide us to have reliable system. GameManagement includes MapManager for updating the map and checking the map. Moreover, it will have SeedManager for checking the growing seed in the land and it will report seed status to the MapManager.

2.3 Architectural Styles

2.3.1 Layers

System of game will be separated to 3 different layers including UI (User Interface), Game-Map management, and game entities. It can be implied that UI system decomposition does not depend from any other subsystem unlike Game Management and Game Entities system decompositions. On the other hand, last layer Game Management subsystem

depends from UI and also controls Game Entities and Game Logic. Additionally, Game Entities subsystem are composed of all entity objects which are brought together.



2.3.2 Model View Controller (MVC)

Aim of this architectural style is to separate your subsystems into 3 parts including controller, model, and view. As it is implied in the layers section we have 3 different layers, which these layers will be classified for MVC. In MVC architectural style, controller responds to user's input by updating model objects. Besides, view is the demonstration of the models in desired format. And model is to management of the objects' data. In relation with our system, Game Management Subsystem establishes Controller. User Interface which is interaction between user and system establishes View. At last, Game Entities Subsystem makes up model of the MVC.

2.3 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 and screen.

2.4 Persistent Data Management

Farmio involves data storage in a filesystem rather than a database. More specifically, data will be stored in "text files" on the hard drive. As the player makes 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.

Rendering the map of the game relies heavily on these data since there will be many blocks of soil (i.e. farm slots) with possibly different seeds at different conditions. Any changes to these farm slots are supposed to be reflected in these text files.

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 about whether to start a new game or load an existing one, the text file(s) to read and write will vary.

Finally, the player's preferences are kept in text files, too.

2.5 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.

2.6 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 to the Pause Menu. Then, after choosing whether to save the game or not, the player will be able to quit the game.

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

In this section we will provide the detailed information about all interfaces of our subsystems and classes.

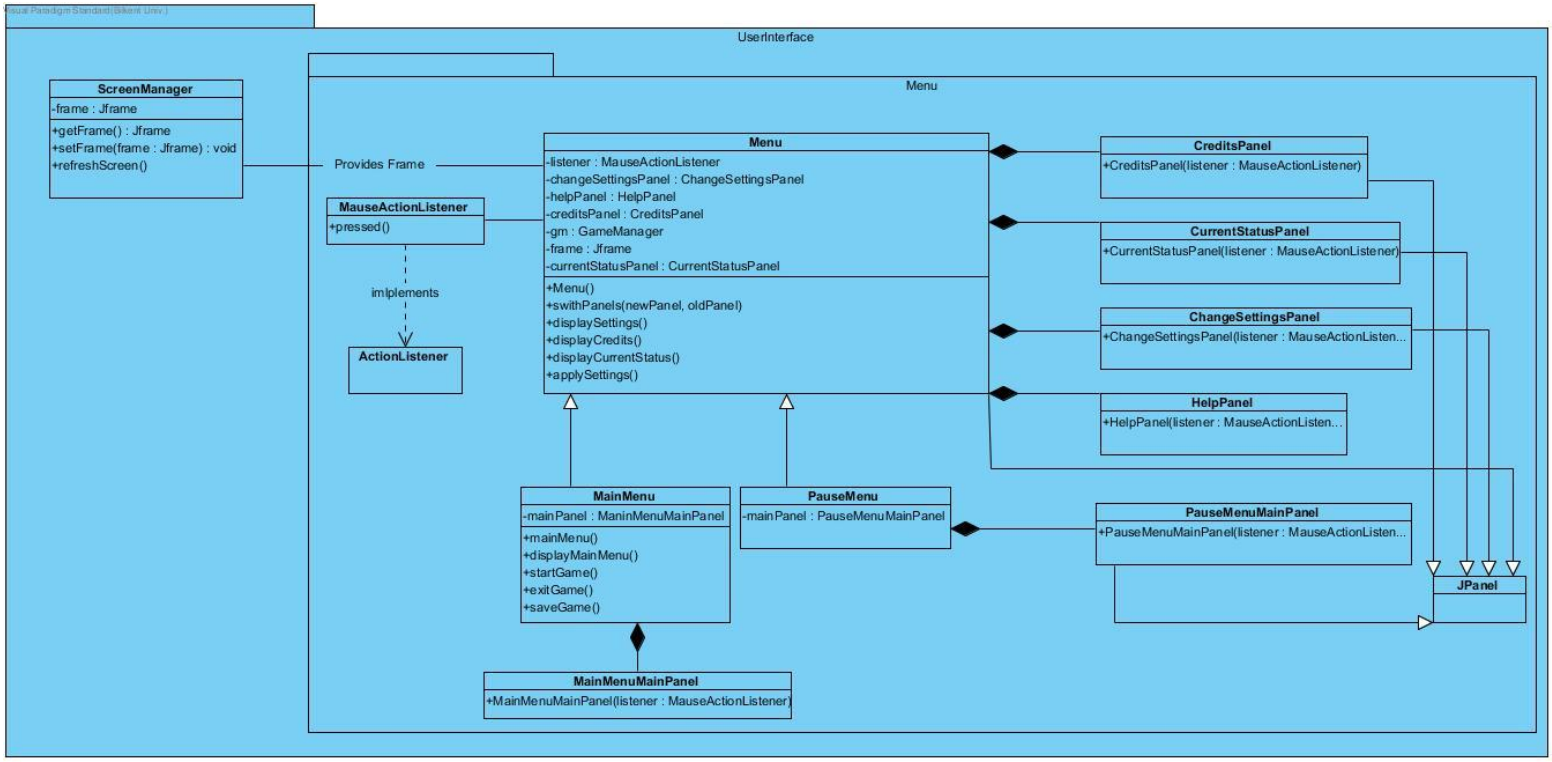
3.1 Design Patterns

Façade Design Pattern:

We are planning to implement our system in Façade design pattern since this structural design pattern helps developers to easily manage subsystem. This design pattern has Façade class which is essential to communicate with classes of other subsystems. Also, Façade class is the only class that communicates with other classes in certain subsystem.

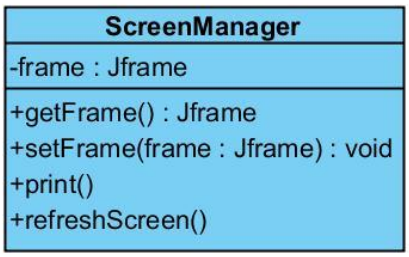
We designed our 2 subsystems in Façade design patterns including, game entities subsystem and game management subsystem. In game management, our Façade class is MapManager class, which controls and communicates with our subsystems and other classes through the actions of the user. Besides, our Façade class in game entities is Map which controlled and modified by the game management subsystem.

3.2 User Interface Subsystem Design



Detailed information about the stated classes in this figure is bellow.

Screen Manager Class

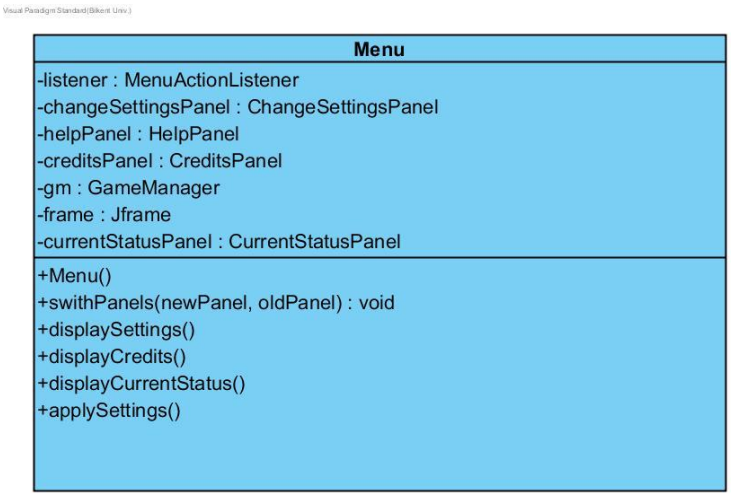


Main action of this class is to provide a screen image for MainMenu. It will be provided by JFrame and it can be changed and refreshed.

Functions

Class Name : ScreenManager	**Explanation of functions**
Public getFrame(): JFrame	Returns the current frame
Private setFrame(frame: JFrame) : void	Sets a new frame.
refreshScreen()	Refreshes the frame after setting one new frame.

Menu Class



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.

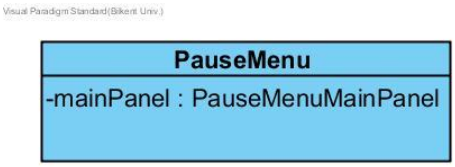
These attributes will be used inside of the functions Menu class. These functions are explained as following :

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

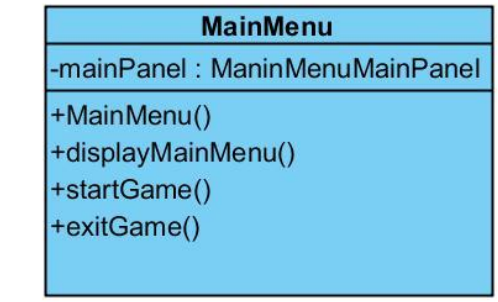


This is the menu that will be able to be accessed from the game page. This Pause Menu will have following functions. This function will connect to the main menu so it's actions will be done.

Functions

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

Main Menu

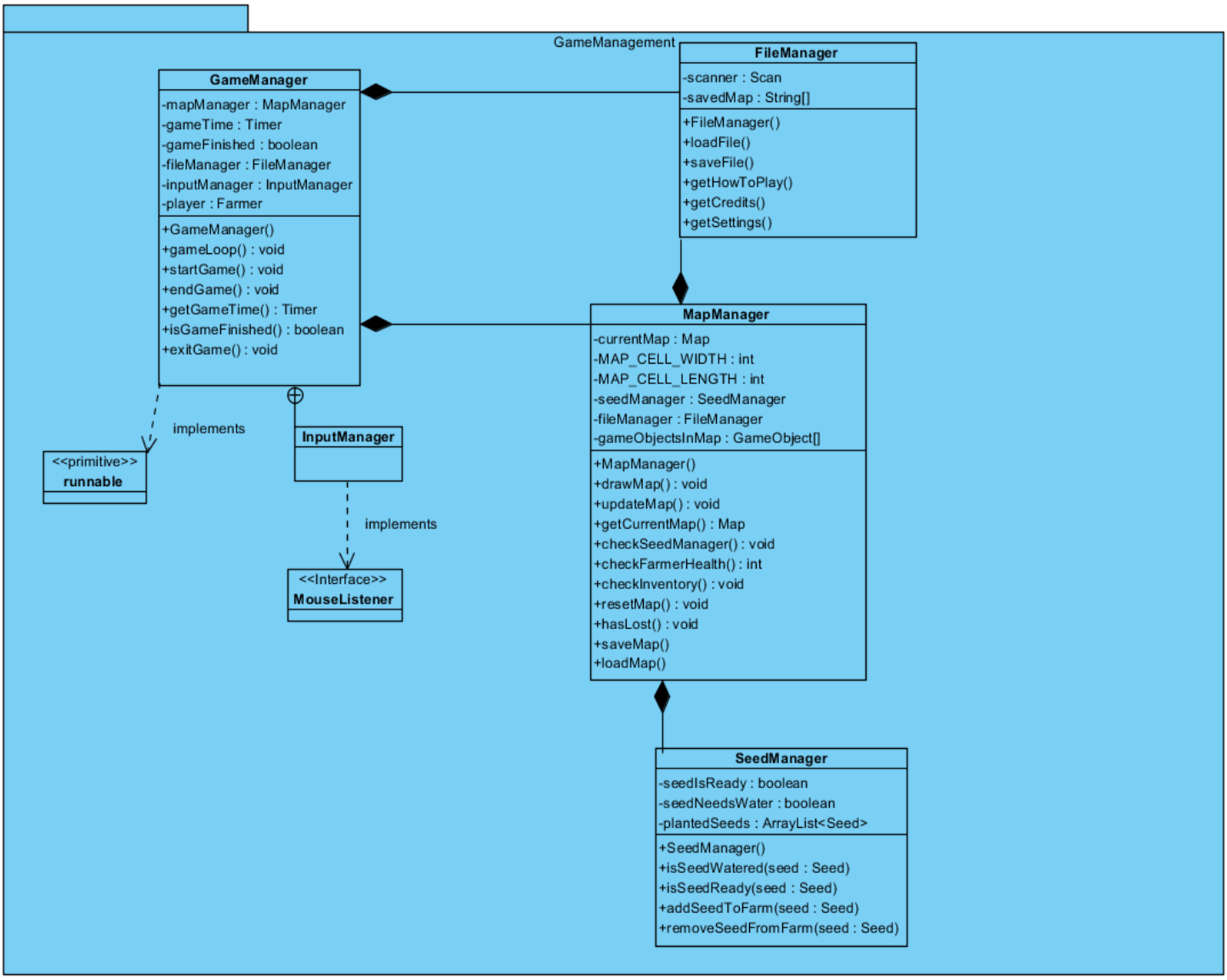


Main menu is the menu that will enable the user to interact with the game with selecting new game or reloading the last saved one, saving the current status of game, exiting game. Main Menu's functions are explained bellow :

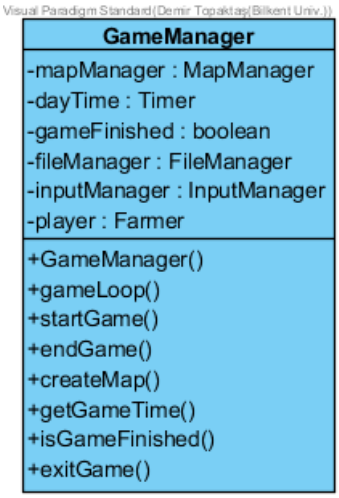
Functions

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.

3.3 Game Management Subsystem



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

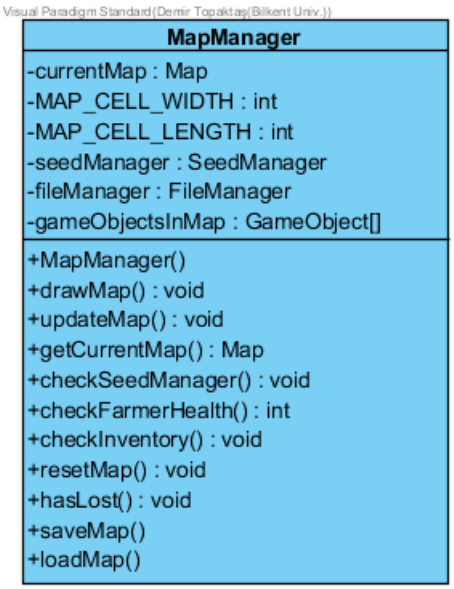
Class Name : Game Manager	Explanation of attributes
private MapManager mapManager:	It is the instance of MapManager. It manages Map and provides the communication between GameManager and Map.
private Timer gameTime	This attribute keeps track of game time and it is initiliazed when the game starts.
private 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

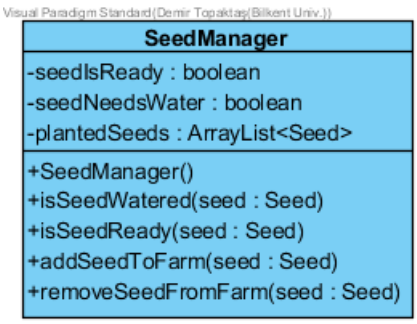
Constructor

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



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.

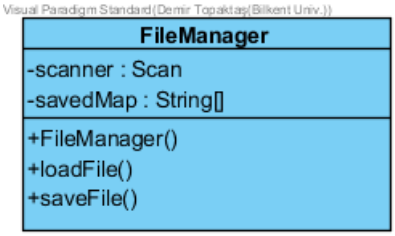
Constructor

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**.**

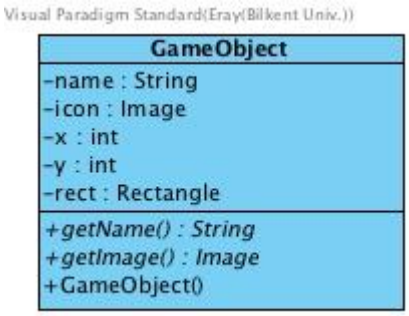
3.4 Game Entities Subsystem

Game Entities Subsystem

Before getting into details of the classes of this subsystem, note that some information is not shown for easier understanding:

- For the sake of clarity, getters and setters are neither mentioned in the descriptions nor shown in the UML diagrams (except the ones whose absence may cause some confusion). However, there will be, of course, getters and setters for all instance variables.
- Unless overridden, inherited attributes are not always mentioned in child classes' properties (again, unless these members are crucial in understanding the rationale).
- Finally, default constructors are not specified in the descriptions.

GameObjects Subsystem



GameObject Class

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

Attributes

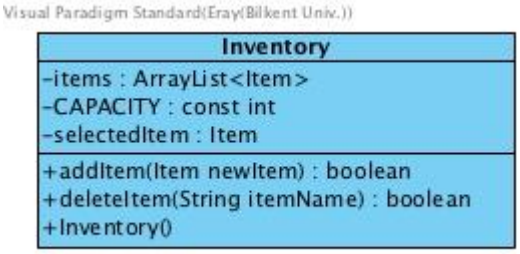
ClassName : GameObjectClass	**Explanation of attributes**
private int x;	is the x-coordinate of a *GameObject* (on the screen).
private int y;	is the y-coordinate of a *GameObject*.
private Rectangle rect	is used to define the boundary areas of the objects (for the game logic).

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



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).

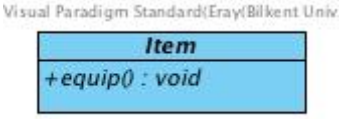
private const int CAPACITY;	defines the maximum number of *Item* instances that can be stored in the inventory.
private Item selectedItem;	is the *Item* instance that is currently chosen / equipped.

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

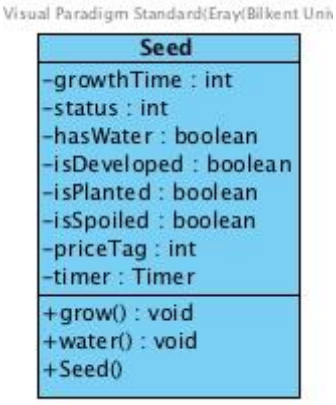


Being another child class of *GameObject*, this abstract class is the parent of *Seed*, *Food* and *WateringCan* classes.

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)

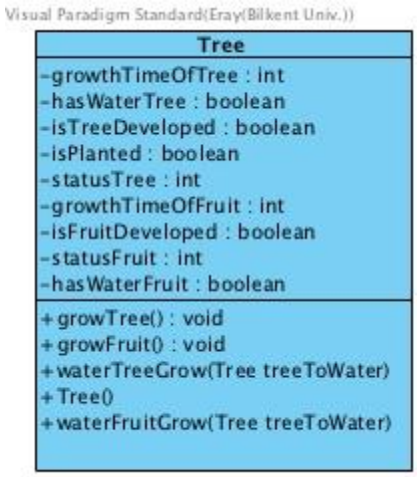
Similar to the *Seed* class, *Food* is a child of *Item*. Moreover, its kinds (*Strawberry*, *Corn*, *Sunflower, Potato, Tomato, Cherry, Raspberry* and *Apple*) are represented as the children of this class.

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

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*.

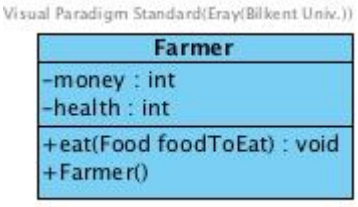
Store Class

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



This class represents the *Farmer*.

Attributes

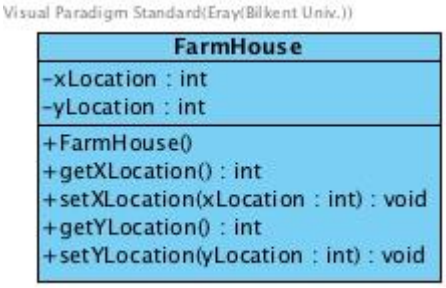
ClassName : FarmerClass	**Explanation of attributes**
private int money;	**is the amount of *money* that the *Farmer* has.**
private int health	represents the *health* level of the *Farmer*.

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



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

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

Attributes

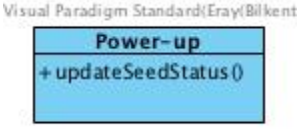
ClassName : MapClass	**Explanation of attributes**
private ArrayList< Grass> availableSlots	This ArrayList is to keep available *Grass* slots to change them into plantable *Pit* objects to plant *Seed* instances.
private ArrayList< GameObject> gameObjects	This list is to keep track of all objects by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

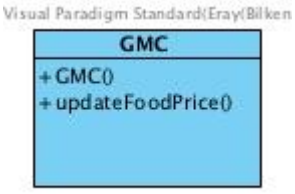
Functions

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

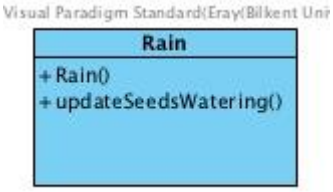


Standing for "Genetically Modified Crop," this is another child of the *Power-up* class. After it is bought from the store, it is applied on *Seed* instances.

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.

4. Conclusion

In this design report, we decided to add new features, classes after analysis report. Firstly, we have added new Entity objects including Potato, Tomato, Cherry, Raspberry, and Apple class as a subclass of Food class. Accordingly, TomatoSeed and PotatoSeed classes have been added. Besides, Tree class added with AppleTree, RaspberryTree, and CherryTree classes, which Tree class is superclass of these classes. In other words, we added ability to plant new trees to our farmland and new type of plants. Also, we added power-ups like GMO, Fertilizer, and Rain. These new features, will increase the entertainment level of game.