
Bilkent University

Department of Computer Engineering

# CS 319 Project

*Project short-name: Settlers of Anatolia*

## Design Report

Enes Merdane, Mehmet Alper Genç, Irmak Demir, Göksü Turan, İrem Kırmacı

Instructor: Uğur Doğrusöz
Teaching Assistant(s): Hasan Balcı

Design Report
November 10, 2019

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Object Oriented Software Engineering course CS319.

# Contents

# Design Report

*Project short-name: Settlers of Anatolia*

## 1. Introduction

### 1.1. Purpose of the system

Settlers of Anatolia is a strategy game inspired by the famous board game Catan. Our aim is to make an entertaining game which also improves the player's strategic thinking. Furthermore, our game will be easy to use and learn, and also modifiable so that we can add new features to the game. The game is won by settling in an island and building constructs, and the first player to reach 10 points wins the game. Our game is a single-player game, and is played with bots.

### 1.2. Design Goals

#### 1.2.1. Ease of Use and Learning

Settlers of Anatolia will be user-friendly and easy to learn. We will include a "How to Play" section that explain the game and its rules in a simple but complete fashion. Furthermore, the user interface of Settlers of Anatolia will be simple and easy to use by everyone.

#### 1.2.2. Runtime Efficiency

Java is a language that spends more time compared to languages like C and C++ [1]. Keeping this in mind, we have to design our project to be as efficient as possible, to minimize the time spent. Furthermore, we believe that games need to be fast and efficient to not lose the interest of the players.

#### 1.2.3. Portability

Since we will be using Java to implement Settlers of Anatolia, the game will be played by any computer that can run the JVM (Java Virtual Machine) and has it installed.

#### 1.2.4. Modifiability

Like many other games, Settlers of Anatolia will also need to be modifiable so that improvements and new features can be added later on. We are aiming to achieve a modifiable project by implementing a three-tier architecture and making our project have as low coupling as possible and as high coherence as possible.

## 2. High-level Software Architecture

### 2.1. Subsystem Decomposition

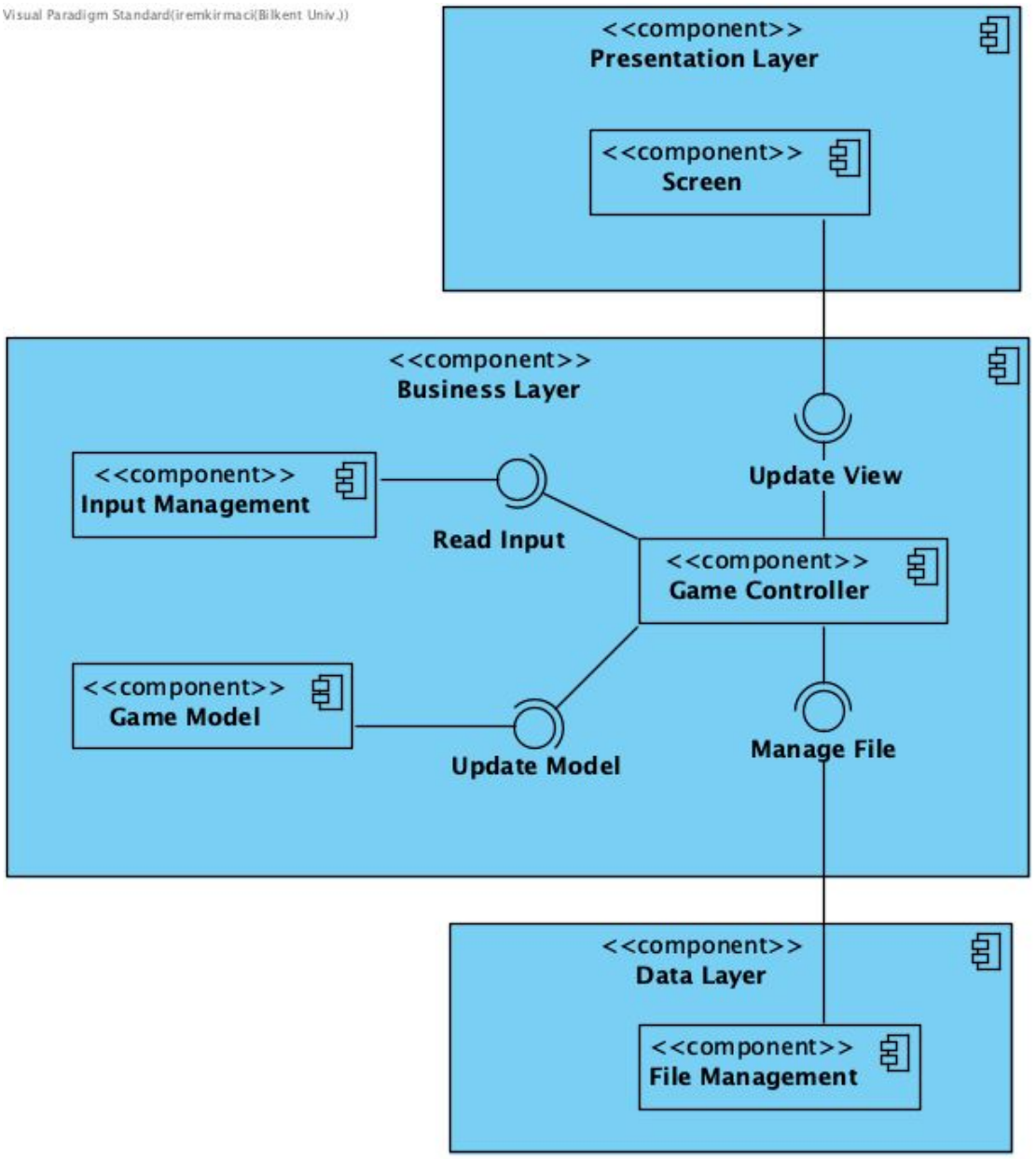



Figure 1: Three Tier Structure and Components

We have chosen to apply three-tier architecture since it is more suitable for our project design and it provides us ease in the implementation part. We are aiming to decrease coupling and increase cohesion as possible. Therefore, we divide our system into 3 different layers which are named Presentation, Business and Data. All these subsystems will have different functions and have interactions with each other. Presentation Layer will provide user interface operations. The interaction with the user will be in this subsystem's responsibility. Business Layer will include Input Management, Game Model and Game Controller operations. Game Controller will receive the inputs from input

management and make modifications on both the Presentation Layer and the Game Model Component. Our another subsystem Data Layer will manage the interaction between file and game for our savings which are settings and current situation of the game. Thus, storing the information in a file base system is another cause of why this architecture is suitable for our design.

### 2.2. Hardware/Software Mapping

As the game will be implemented using JDK 8, which is developed by Oracle. We have decided to use JDK 8 since it has JavaFX in its own libraries. More recent versions of the JDK do not include JavaFX. The executable file of the game will be an executable jar file [2].

Thus, the players will need to have a Java Runtime Environment (version 8u231), which supports JavaFX projects, installed on their computers [3]. We will use Maven, a build management tool, for all sorts of tasks that are required to build our project [4].

For the hardware specification, the user will need a Java runtime environment supported operating system installed on their computer, a keyboard and a mouse to play the game. Also, to listen to the game music and to hear sound effects, they will need a headphone or speaker connected to their computer.

### 2.3. Persistent Data Management

The background image of the main menu and the game screen, the pictures of sources and player avatars will be kept in the .jpeg format in the file system. However, the rest of the components will be hardcoded so they will be drawn during the execution of the game. By hardcoding some of the visual components, we aim to reduce the memory space that will be allocated for the game. We will also be storing our audio files in the wav format in the file system.

There will be file management system in the game to store necessary game objects in order to save the information of the latest played game and the settings preferences of the player to use when it is needed. This file management system will use ObjectInputStream and ObjectOutputStream classes to perform its tasks.

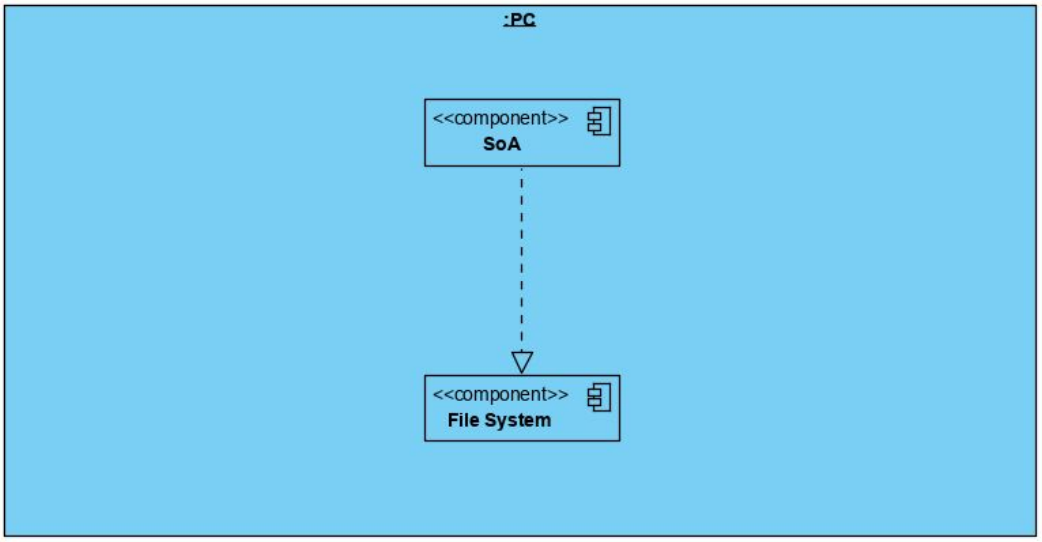



Figure 2: Deployment Diagram

### 2.4. Access Control and Security

Settlers of Anatolia game does not collect any kind of data of the users. It only takes the username to display in the game and when the game is terminated, this information will not be kept anywhere.

Also, in the Settlers of Anatolia, there is no need to be connected to the Internet because the game is not using any services or database which needs Internet connection. Thus, there will not be any security issues related to this.

| | GameController | GameModel |
|---|---|---|
| Player | startGame()<br>pauseGame()<br>exitGame() | trade()<br>playCard()<br>buyProgressCard()<br>build()<br>playTurn() |

Figure 3: User Access to Classes Matrix

In terms of the system, as it is seen in the given table, the methods the player can access are very limited and has the operations only related to the playing of the game so the player cannot change anything related to the game system.

### 2.5. Boundary Conditions

#### 2.5.1. Initialization

The settlers of Anatolia will use the executable ".jar" file therefore it will not require any installation for the game. It will not have multiplayer feature so it will not require to connect to any server database.

#### 2.5.2. Termination

The game could be terminated only by clicking the "Exit" button in the main menu. Although the game has already started, for termination the user has to go back to main menu with "Back to Main" button and then terminate the game from "Exit" button.

### 2.5.3. Errors

If an error occurs during the game, the game will not be stopped. It will continue with the next step. If an error occurs while reading the previous game from a file, the game will report to the user and create a new game. If an error occurs while reading the settings from a file, the game will instead load the default settings.

## 3. Subsystem Services

### 3.1. Presentation Layer

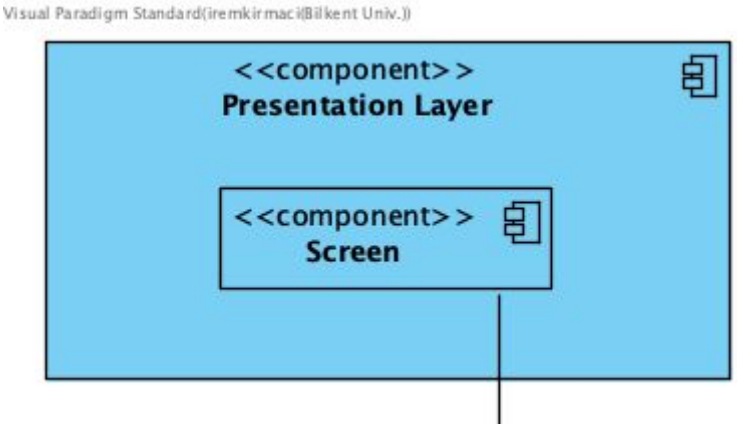



Figure 4: Presentation Layer

Presentation Layer presents the user interface for Settlers of Anatolia with one component which is Screen.

When the Screen component is invoked with update view, proper screen will be displayed. These screens could be Main Menu, How To Play, Settings, Game Options, Game Menu, Trade, EndGame, or GameScreen. Screen component make the proper updates on the screen.

### 3.2. Business Layer

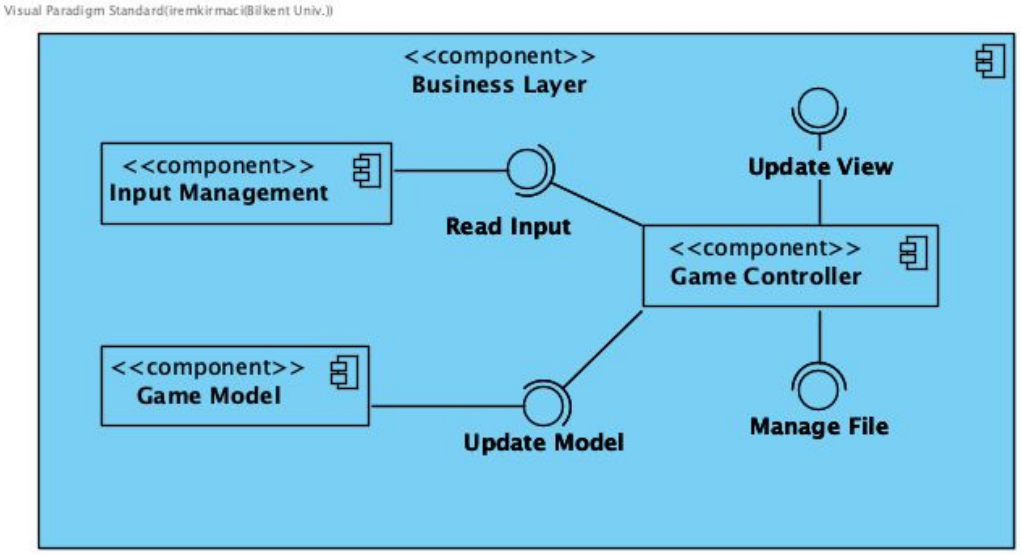



Figure 5: Business Layer - High Level Design

In Business Layer, there are 3 major components named Input Management, Game Model and Game Controller. The input of the interaction will be taken from Input Management and provided to the Game Controller. Based on this input, Game Controller will be evoked and it determines updates and controls the states. Game Model component provides these updates with the required objects. Changes on the map which are done by Game Model are controlled and updates are evoked by the Game Controller component with respect to them. Furthermore, when the player exits the game, the last updated version of the game and the settings preferences will be saved to Data Layer.

### 3.3. Data Layer

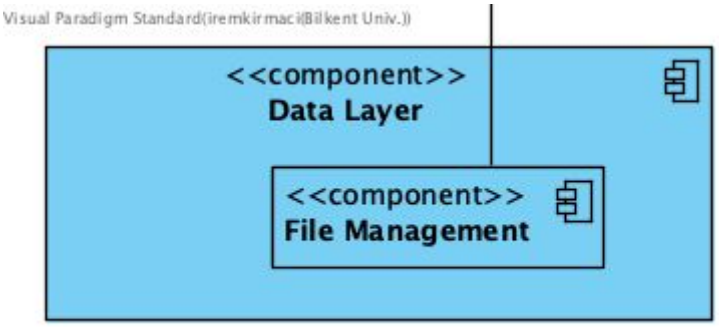



Figure 6: Data Layer - High Level Design

The Data Layer subsystem will be used to write certain objects to files and read objects from files. Since our game saves the current situation of the game, its settings and continuing later on, the File Management will be evoked with respect to Game Management's decision.

## 4. Low-level Design

### 4.1. Object Design Trade-offs

#### 4.1.1. Ease of Use vs. Functionality

Due to the fact that one of our design goals is usability and understandability of the game, ease of use has the higher priority against functionality. Generally, as the number of functions increase in one user interface, the understandability and learnability of interfaces decrease. Therefore, as our design goal we are planning to focus more on ease of use.

#### 4.1.2. Memory vs. Performance

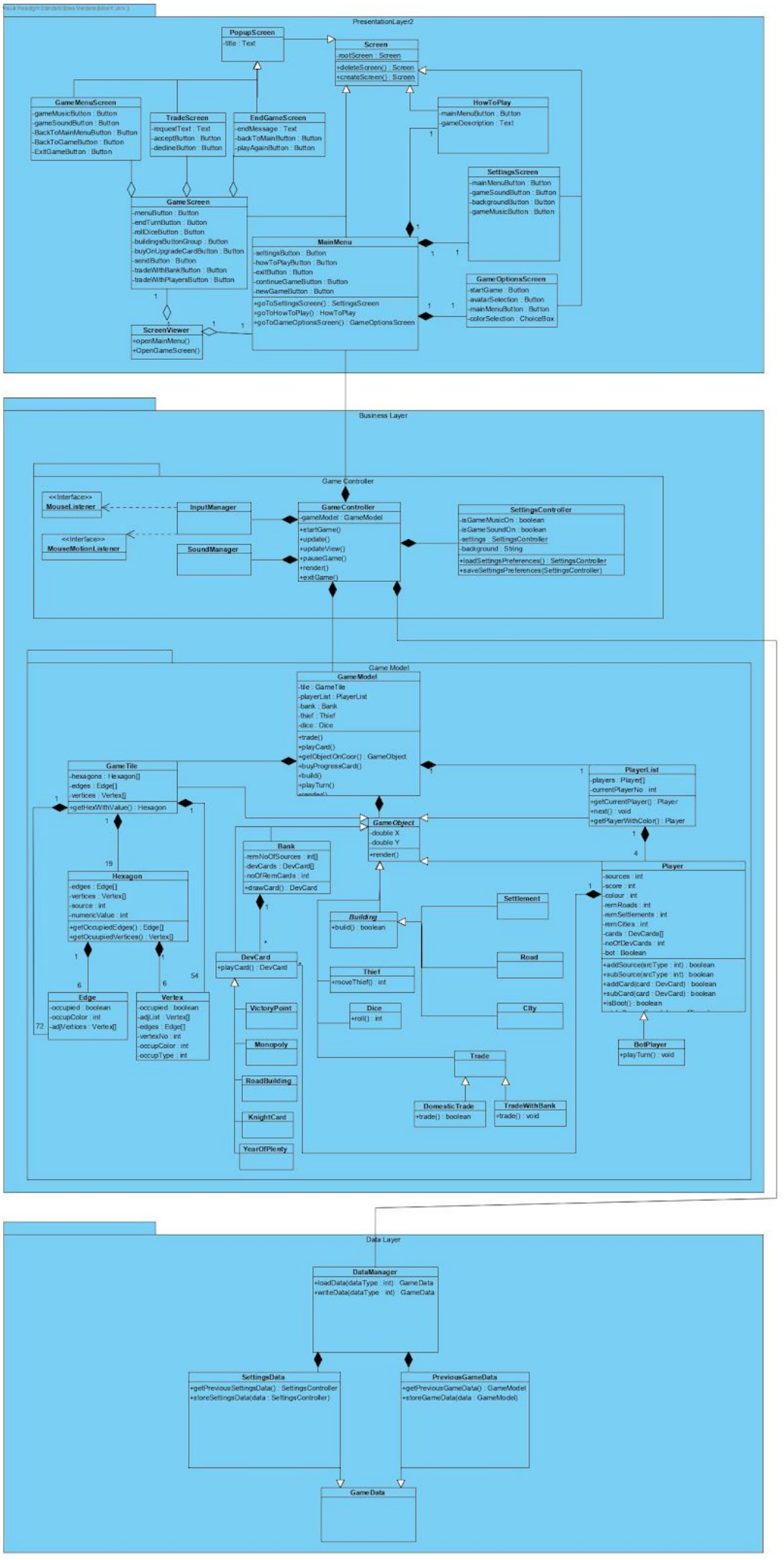
We believe that the game must be fast and efficient. Also, we do not want our players to lose interest waiting for the game to process their actions or for loading screens to finish. This is why we will use more RAM storage to keep references so that the instances have quicker access to other subsystems. Therefore, we prefer performance to memory.

#### 4.1.3. Modifiability vs. Robustness

As our design goals suggest, modifiability is an important feature of our project because we will add new features in the following iterations and we need to do these changes easily. However, while making these changes, it is important that the main content of the system should remain as it is. Therefore, robustness is important as well. However, although we try to make them both, modifiability is more essential for us. Thus, we will focus more on modifiability.

### 4.2. Final Object Design

The complete diagram of the Final Object design is in the next page.


PresentationLayer2
PopupScreen
-title : Text
Screen
-rootScreen : Screen
+deleteScreen() : Screen
+createScreen() : Screen
GameMenuScreen
-gameMusicButton : Button
-gameSoundButton : Button
-BackToMainMenuButton : Button
-BackToGameButton : Button
-ExitGameButton : Button
TradeScreen
-requestText : Text
-acceptButton : Button
-declineButton : Button
EndGameScreen
-endMessage : Text
-backToMainButton : Button
-playAgainButton : Button
HowToPlay
-mainMenuButton : Button
-gameDescription : Text
SettingsScreen
-mainMenuButton : Button
-gameSoundButton : Button
-backgroundButton : Button
-gameMusicButton : Button
GameScreen
-menuButton : Button
-endTurnButton : Button
-rollDiceButton : Button
-buildingsButtonGroup : Button
-buyOrUpgradeCardButton : Button
-sendButton : Button
-tradeWithBankButton : Button
-tradeWithPlayersButton : Button
MainMenu
-settingsButton : Button
-howToPlayButton : Button
-exitButton : Button
-continueGameButton : Button
-newGameButton : Button
+goToSettingsScreen() : SettingsScreen
+goToHowToPlay() : HowToPlay
+goToGameOptionsScreen() : GameOptionsScreen
GameOptionsScreen
-startGame : Button
-avatarSelection : Button
-mainMenuButton : Button
-colorSelection : ChoiceBox
ScreenViewer
+openMainMenu()
+OpenGameScreen()
Business Layer
Game Controller
<<Interface>>
MouseListener
InputManager
<<Interface>>
MouseMotionListener
SoundManager
GameController
-gameModel : GameModel
+startGame()
+update()
+updateView()
+pauseGame()
+render()
+exitGame()
SettingsController
-isGameMusicOn : boolean
-isGameSoundOn : boolean
-settings : SettingsController
-background : String
+loadSettingsPreferences() : SettingsController
+saveSettingsPreferences(SettingsController)
Game Model
GameModel
-tile : GameTile
-playerList : PlayerList
-bank : Bank
-thief : Thief
-dice : Dice
+trade()
+playCard()
+getObjectOnCoor() : GameObject
+buyProgressCard()
+build()
+playTurn()
GameTile
-hexagons : Hexagon[]
-edges : Edge[]
-vertices : Vertex[]
+getHexWithValue() : Hexagon
PlayerList
-players : Player[]
-currentPlayerNo : int
+getCurrentPlayer() : Player
+next() : void
+getPlayerWithColor() : Player
GameObject
-double X
-double Y
+render()
Bank
-remNoOfSources : int[]
-devCards : DevCard[]
-noOfRemCards : int
+drawCard() : DevCard
Player
-sources : int
-score : int
-colour : int
-remRoads : int
-remSettlements : int
-remCities : int
-cards : DevCards[]
-noOfDevCards : int
-bot : Boolean
+addSource(srcType : int) : boolean
+subSource(srcType : int) : boolean
+addCard(card : DevCard) : boolean
+subCard(card : DevCard) : boolean
+isBoot() : boolean
Hexagon
-edges : Edge[]
-vertices : Vertex[]
-source : int
-numericValue : int
+getOccupiedEdges() : Edge[]
+getOccupiedVertices() : Vertex[]
Building
+build() : boolean
Settlement
Road
City
Thief
+moveThief() : int
Dice
+roll() : int
DevCard
+playCard() : DevCard
VictoryPoint
Monopoly
RoadBuilding
KnightCard
YearOfPlenty
Edge
-occupied : boolean
-occupColor : int
-adjVertices : Vertex[]
Vertex
-occupied : boolean
-adjList : Vertex[]
-edges : Edge[]
-vertexNo : int
-occupColor : int
-occupType : int
Trade
DomesticTrade
+trade() : boolean
TradeWithBank
+trade() : void
BotPlayer
+playTurn() : void
Data Layer
DataManager
+loadData(dataType : int) : GameData
+writeData(dataType : int) : GameData
SettingsData
+getPreviousSettingsData() : SettingsController
+storeSettingsData(data : SettingsController)
PreviousGameData
+getPreviousGameData() : GameModel
+storeGameData(data : GameModel)
GameData

### 4.2.1. Presentation Layer

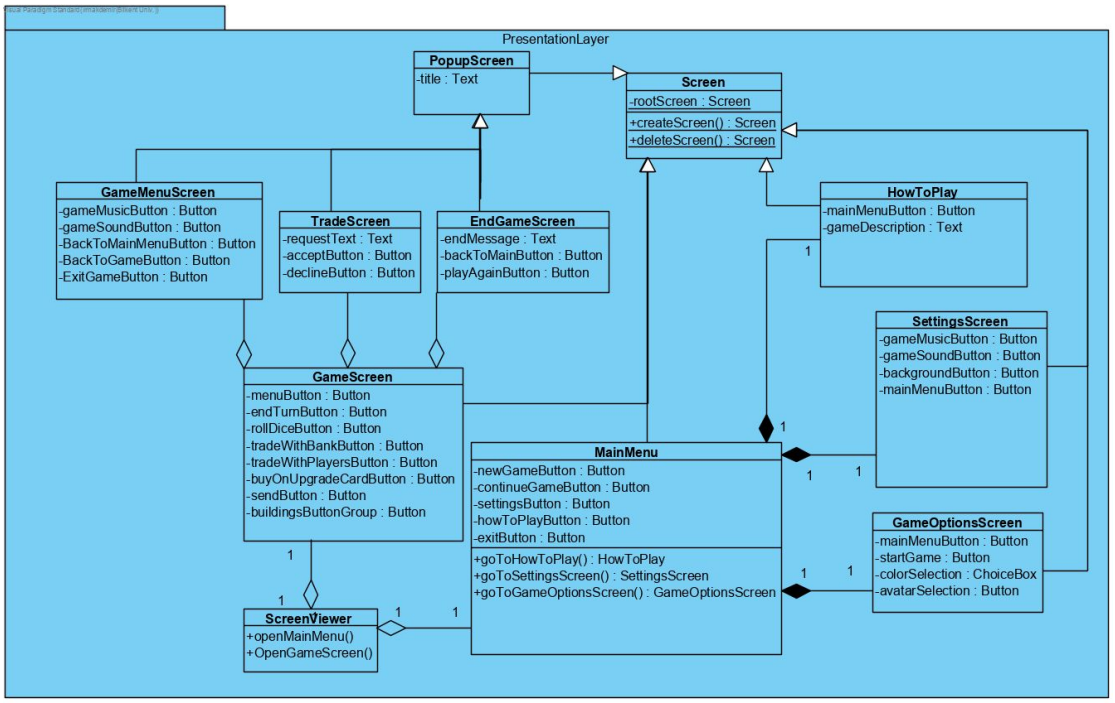



Figure 8: Presentation Layer Classes

#### 4.2.1.1. ScreenViewer


Figure 9: ScreenViewer Class

This is the Façade class for the Presentation Layer, meaning that other classes call one of ScreenViewer's methods to interact with any class in the Presentation Layer. The ScreenViewer class has `openMainMenu()` method for opening the Main Menu, and `openGameScreen()` method for opening the Game Screen.

#### 4.2.1.2. Screen

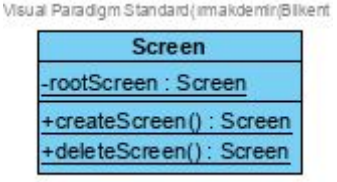



Figure 10: Screen Class

The Screen class is an abstract class that is extended by every class in the Presentation Layer other than ScreenViewer. Since all Screen-related classes are Singleton objects, the Screen class contains two abstract methods `createScreen()` and `deleteScreen()` that are later realized.

#### 4.2.1.3. PopupScreen

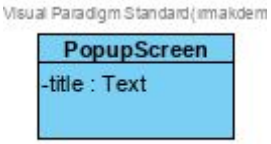



Figure 11: PopupScreen Class

PopupScreen is an abstract class for any kind of popup that can appear on the game screen. This class is realized by the classes GameMenuScreen, TradeScreen and EndGameScreen.

#### 4.2.1.4. GameMenuScreen

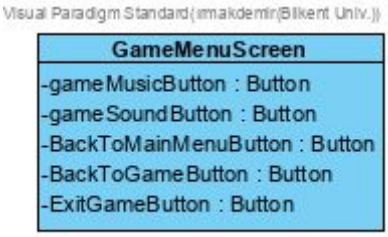



Figure 12: GameMenuScreen Class

This class represents the screen that pops up when the user presses “Esc” on their keyboard or clicks the “Menu” button while playing the game. This screen contains buttons to edit the music and sound volume, go back to main menu, exit game, and close the menu and continue playing.

#### 4.2.1.5. TradeScreen

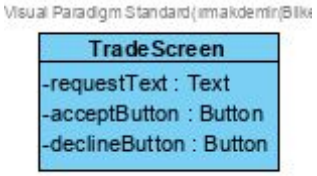



Figure 13: TradeScreen Class

This class represents the screen that pops up when the user receives a trade request from another player. It contains a Text object describing the contents of the trade request, and buttons for accepting and declining the trade request.

#### 4.2.1.6. EndGameScreen


Figure 14: EndGamseScreen Class

This class represents the screen that pops up when the game ends. It contains an end message, either telling the player that they have won or that they lost, in which case the name of the winning player is also included. This screen also contains two buttons: One for going to Main Menu, and another for playing again.

#### 4.2.1.7. GameScreen


Figure 15: GameScreen Class

This class represents the main display of the game, which contains the elements of the board game, such as the board and cards. There are 8 Button objects in this class, whose tasks are opening the menu, ending the turn, rolling the dice, trading with bank, creating trade request, buying upgrade cards, sending the created trade request and building a construct on the map, respectively.

#### 4.2.1.8. MainMenu

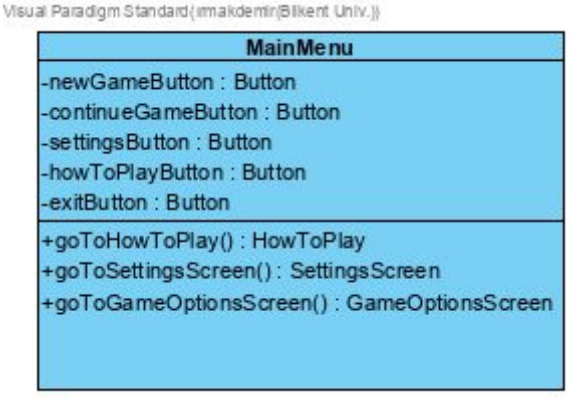



Figure 16: MainMenu Class

This class represents the Main Menu of the game, which opens when the game is initiated. This class contains 5 buttons, which are for starting a new game, continuing the previous game, going to the “Settings” screen, going to the “How to Play” screen and exiting the game. There are `goToHowToPlay()`, `goToSettingsScreen()` and `goToGameOptionsScreen()` methods for going to the mentioned screens.

#### 4.2.1.9. HowToPlay


Figure 17: HowToPlay Class

This is the class for the screen that opens when the user clicks “How to Play” button while in the Main Menu. This screen contains general information about the game in a Text object. There is also a button that leads the user back to the Main Menu.

#### 4.2.1.10. SettingsScreen

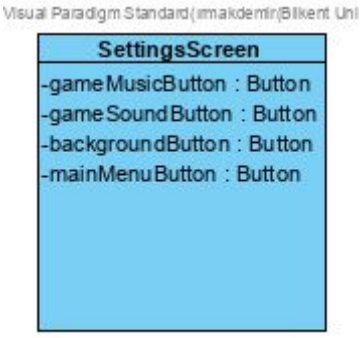



Figure 18: SettingsScreen Class

This is the class for the screen in which the user can edit the settings. Options include changing the volume of the game sounds and music, changing the background image and returning to the Main Menu.

#### 4.2.1.11. GameOptionsScreen


Figure 19: GameOptionsScreen Class

This class represents the screen which is opened when the player selects the “New Game” option, before the game starts. This class contains a choice box for color selection, and three buttons: One for returning to the Main Menu, another for avatar selection, and another for returning to the Main Menu.

### 4.2.2. Business Layer

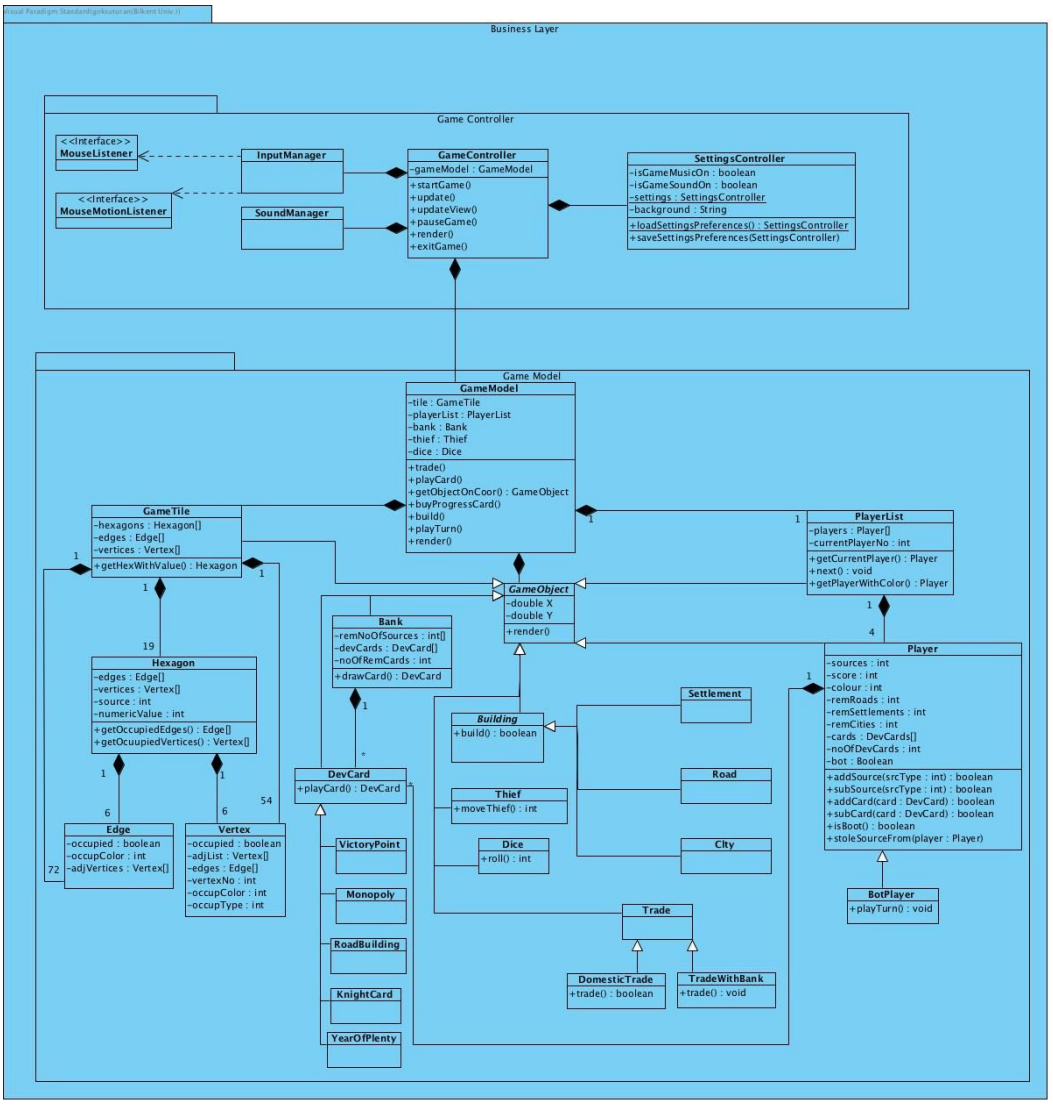



Figure 20: Business Layer Classes

#### 4.2.2.1. Game Controller Package

##### 4.2.2.1.1. GameController


Figure 21: GameController Class

The GameController class, as its name suggests, controls the state of the game and responds to the actions of the user. It holds a GameModel instance to be able to control the game. It has 5 methods: a startGame() method, an `update()` method that updates the GameModel based on the user's actions, and `updateView()` method that updates the GUI, a `pauseGame()` method and a `render()` method that calls each GameObject's `render()` method for the GUI to be able to draw them.

##### 4.2.2.1.2. SettingsController


Figure 22: SettingsController Class

The SettingsController class holds the settings preferences of the user and changes the game accordingly. It is a Singleton class, and the single instance is obtained by the `loadSettingPreferences()` method. There are two boolean properties that hold whether or not the user wants the game music and sounds on or off. As Singleton pattern requires, there is a SettingsController object as a property. There is also a String named "background" which holds the name of the background image. This String can not be manipulated by the user, but the user can select from a number of pre-defined image names. Finally, there is a `saveSettingsPreferences(SettingsController)` method that saves the user's settings preferences into a file.

#### 4.2.2.1.3. InputManager

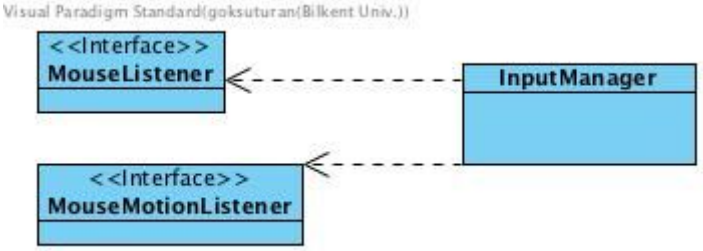



Figure 23: InputManager Class

The InputManager class regulates the keyboard and mouse inputs of the user, and tells the GameController what must be done.

#### 4.2.2.1.4. SoundManager


Figure 24: SoundManager Class

The SoundManager class is responsible of playing the sounds and background music, according to the player's preferences. If the player does not want the music and/or the sounds to play, then the SoundManager will not play them.

### 4.2.2.2. Game Model Package

#### 4.2.2.2.1. GameModel


Figure 25: GameModel Class

GameModel class regulates the inner interface of the game. It has objects of classes GameTile, PlayerList, Bank, Thief and Dice. According to the GameController's instructions, GameModel calls the related methods in which the game objects' methods are called, to draw game objects, create trade request and trade, to roll the rice, to play or buy some cards and in order for a particular player to play its turn.

#### 4.2.2.2.2. GameObject

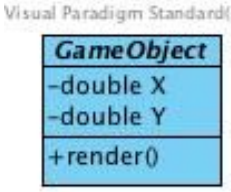



Figure 26: GameObject Class

GameObject class is an abstract class which ensures that every game object such as Building, Bank, is locatable with x and y coordinates and renderable.

#### 4.2.2.2.3. GameTile


Figure 27: GameTile Class

GameTile class represents the map of the game. It consists of an array of 19 hexagons having different numeric values. Game tile has edge and vertex list within itself which is inspired from graph representation, build into the map conveniently.

#### 4.2.2.2.4. Hexagon

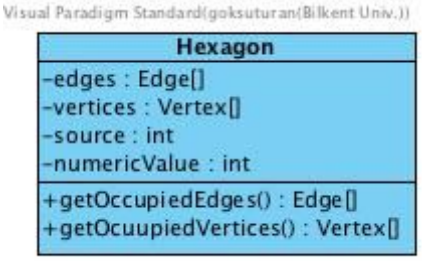



Figure 28: Hexagon Class

Hexagon class regulates the building behaviour. With the list of edges and vertices within itself, hexagon can get each vertex and edge occupied in two arrays. Hereby, the sources can be distributed to the owners of the buildings at each vertex or edge.

#### 4.2.2.2.5. Edge


Figure 29: Edge Class

Edge has an attribute called `occupied` to check if there is a road in that particular edge. It has also an attribute called `occupColor` to get the color of the road build if the edge is occupied. By the `occupColor` attribute the player who is the owner can be identified. Inspired by the graph structure, the list of adjacent vertices is kept to check constructability of the road into that edge.

#### 4.2.2.2.6. Vertex


Figure 30: Vertex Class

Vertex has an attribute called `occupied` to check if there is a settlement in that particular vertex. It has also an attribute called `occupColor` to get color of the settlement build if the vertex is occupied.By the `occupColor` attribute the player who is the owner can be identified. By the `occupType` attribute, that if the building at particular vertex is a settlement or city is identified. Inspired by the graph structure, the list of adjacent vertices and edges are kept to check constructability of the settlement into that vertex.

#### 4.2.2.2.7. Building


Figure 31: Building Class

Building class is an abstract class which ensures that Road, City and Settlement are buildable game objects.

#### 4.2.2.2.8. Settlement - Road - City

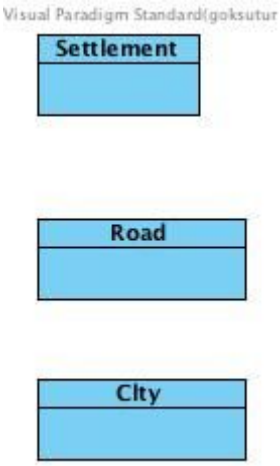



Figure 32: Building Classes

Settlement, Road and City classes are the specialization classes of the Building class. These are buildable GameObjects.

#### 4.2.2.2.9. Bank


Figure 33: Bank Class

The Bank class represents the bank of our game which the players can trade resources with and draw a card from. the remNoOfSources integer array contains the remaining resources in the bank. `devCards` holds the deck of development cards that the players can draw from. `noOfRemCards` holds the number of remaining cards in the bank. It has a method `drawCard()` that returns a card from `devCards` and removes that card from the array.

#### 4.2.2.2.10. DevCard

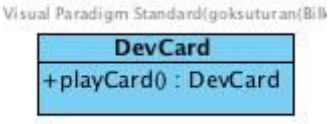



Figure 34: DevCard Class

The DevCard class is an abstract class that represents the development cards of our game. It has a method `playCard()` that returns DevCard, which applies the actions of the development card.

#### 4.2.2.2.11. VictoryPoint, Monopoly, RoadBuilding, KnightCard, YearOfPlenty

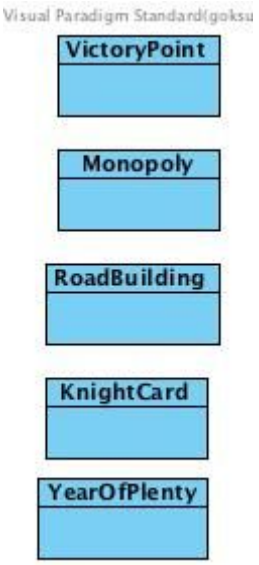



Figure 35: Development Card Classes

VictoryPoint, Monopoly, RoadBuilding, KnightCard and YearOfPlenty classes are the specialization classes of the DevCard class. Each of these are also “GameObject”s.

#### 4.2.2.2.12. Dice

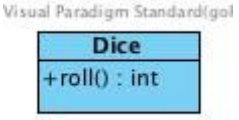



Figure 36: Dice Class

Dice class has an attribute called `roll` to return result of dice. It is also a GameObject.

#### 4.2.2.2.13. Trade, DomesticTrade, TradeWithBank

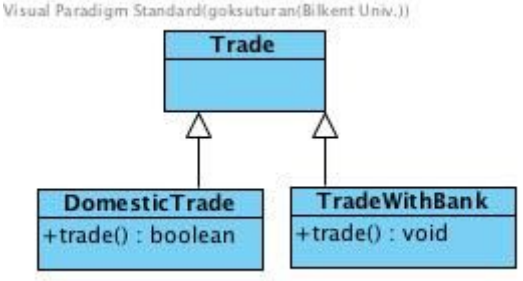



Figure 37: Trade Classes

The Trade class is an abstract class. It is relized by two classes: DomesticTrade and TradeWithBank. DomesticTrade represents trading with another player. It has a `trade()` method that returns whether or not the trade is accepted with a boolean. TradeWithBank represents trading

with the bank. It has a `trade()` method that does not return any value, since the bank does not decline any trade requests, as long as it fits the 4 for 1 convention.

#### 4.2.2.2.14. PlayerList


Figure 38: PlayerList Class

PlayerList class represents the players of the game. It has an attribute called `players` which holds all player instances initialized in the game in an array. Additionally, after each player has played with the `next` operation, turn of play will be passed to the next player. Also by the `getCurrentPlayer` operation it will be reach that which player is the current one.

#### 4.2.2.2.15. Player


Figure 39: Player Class

As the name indicates, Player class is a representative of the user's capabilities and features. About the features of the player, the class has an array attribute called `sources` whose elements hold the number of a particular source that the player owns. For example `sources[0]` will hold the number of woods that the particular player has. It has also another array attribute called `cards`, which holds the Card object instances that a player has, to represents the cards that the player owns. Additionally, player class has attributes for holding the game colour and the score of the player. Attributes called remRoads, remSettlements, remCities in the Player class, will be used to identify the remaining number of building that a player can build into the map and the attribute called `bot` will be used to identify if the particular player instance is a bot.

About the capabilities of the player, player can add or subtract a particular source ( which is hold as an integer )to its sources by the methods `addSource(srcType: int)` and `subSource(srcType: int)`. Similarly, the player can add or subtract a particular card to its deck of development cards by the methods `addCard( card: DevCard)` and `subCard( card: DevCard)`. Player can also stole source from a particular player with `stoleSourceFrom( player: Player)` method. Lastly, `isBoot()` method will be used to identify if a particular player is a bot player.

#### 4.2.2.2.16. BotPlayer

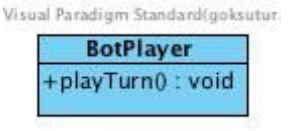



Figure 40: BotPlayer Class

BotPlayer class is a specialization class of the Player class. This class has an operation called `playTurn()` to decide movements of bot players in their turns.

### 4.2.3. Data Layer

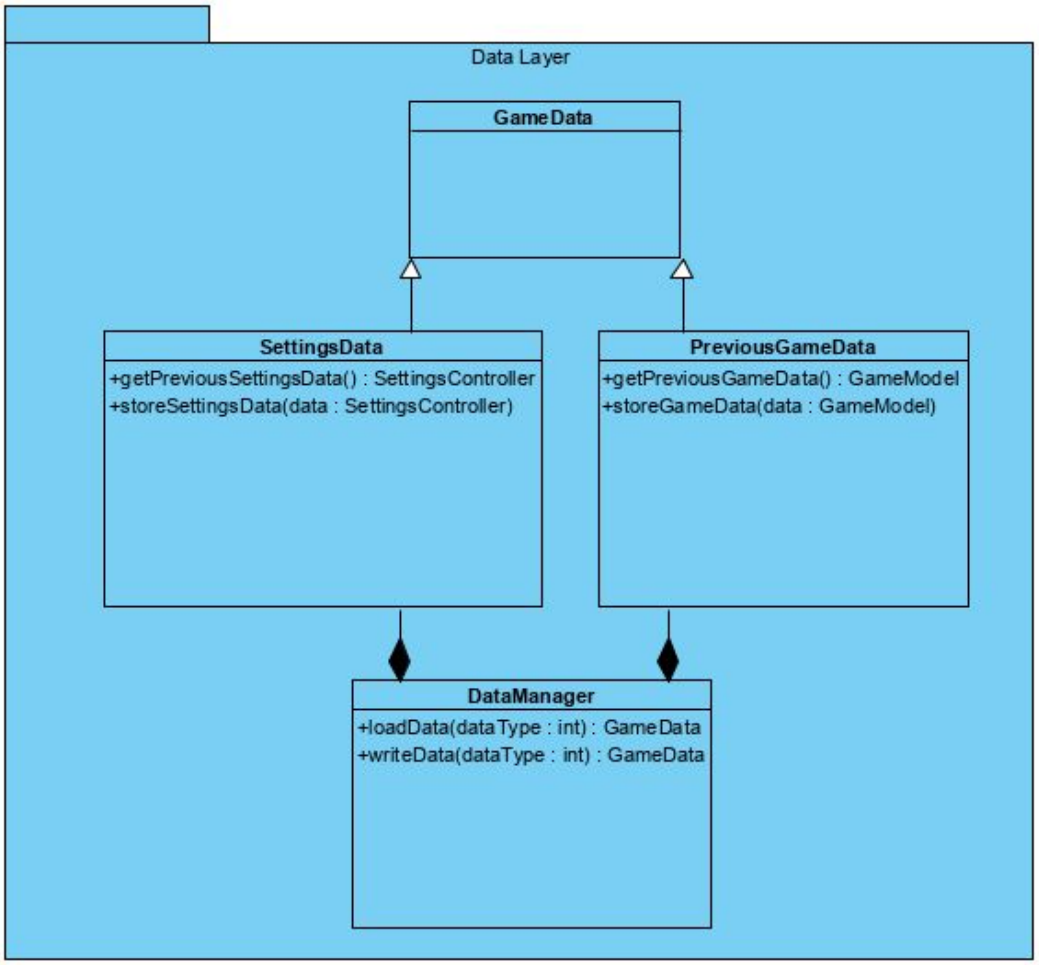



Figure 41: Data Layer Classes

#### 4.2.3.1. GameData


Figure 42: GameData Class

GameData is an abstract class which does not include any properties or methods, it just generalizes the return type of the data. It will be extended by the SettingsData and PreviousGameData classes.

#### 4.2.3.2. SettingsData

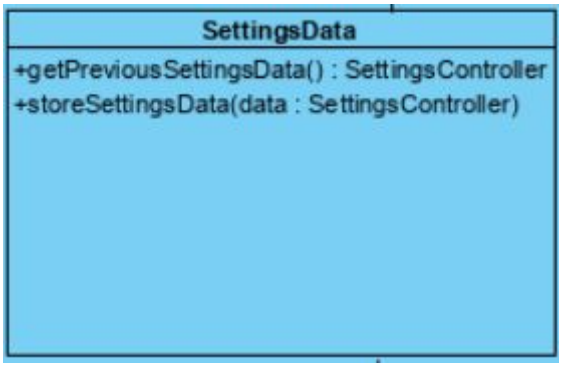



Figure 43: SettingsData Class

SettingsData is the class to load settings preferences of the user from the file system. It has two methods which are `getPreviousSettingsData():SettingsController` and `storeSettingsData(data : SettingsController)`. First method includes operations to get stored SettingsController object from the file system and returns the object to be used. The second method gets a SettingsControler instance and stores it into the file system.

#### 4.2.3.3. PreviousGameData

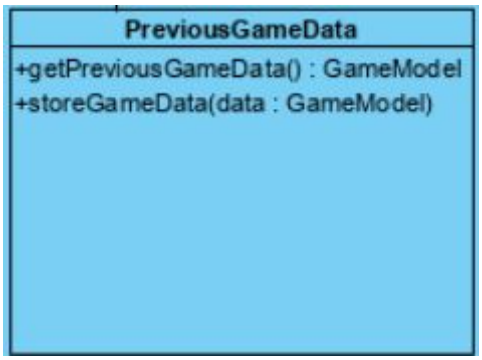



Figure 44: PreviousGameData Class

PreviousGameData is the class to load previous game data from the file system. It has two methods which are `getPreviousGameData():GameModel` and `storeGameData(data : GameModel)`. First method includes operations to get stored GameModel object from the file system and returns the object to be used. The second method gets a GameModel instance and stores it into the file system.

#### 4.2.3.4. DataManager

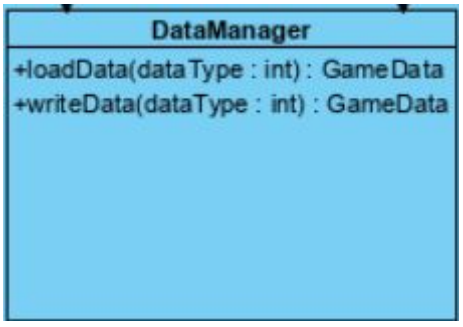



Figure 45: DataManager Class

DataManager is the class used by GameController class in the Business Layer. Through this class, GameController will retrieve the data from the filesystem. It has two methods which are `loadData(dataType : int):GameData` and `storeData(dataType : int, data:GameData)`. The first method will be used to load data from the file system. The caller function will give the parameter 0 for SettingsData or 1 for PreviousGameData. According to this parameter, this method will return either SettingsData or PreviousGameData. The second method is to store data to the file system. It takes two parameters. “dataType” is used as it was in the first method, and the parameter GameData includes the data sent by the caller function to store to the file system.

### 4.3. Packages

The following information in this subsection and subsection 4.4 about the contents of the outside packages, classes and interfaces are from the Java 8 documents [5] and the JavaFX 8 documents [6].

#### 4.3.1. java.util

The java.util package contains very useful classes such as data collections and iterators. We will be using the List class in our implementation, and we will also be using the EventListener interface while handling events.

#### 4.3.2. java.io

We will be using an ObjectInputStream object to read classes from files, and an ObjectOutputStream object to write them to files. Furthermore, to do these operations, the classes that will be written to files need to implement the Serializable interface, which is also in java.io.

#### 4.3.3. java.lang

The classes and subpackages of java.lang are essential for Java programs. Some important examples of java.lang classes in our application are Strings and Exceptions.

#### 4.3.4. javafx.animation

We will use classes from this package to code the necessary animations.

#### 4.3.5. javafx.application

This package contains the Application class, which JavaFX applications need to extend and override its `start(javafx.stage.Stage)` method. This is why we will need this package.

#### 4.3.6. java.css

JavaFX supports styling with CSS, which is one of the reasons we picked JavaFX for our project. We will use CSS code for the styling of our GUIs.

#### 4.3.7. javafx.event

We will be using javafx.event classes to receive the mouse and keyboard actions of the user, such as button clicks or key presses, and make our application respond accordingly.

#### 4.3.8. javafx.scene

JavaFX scenes are essential for JavaFX GUI applications, and we will be using javafx.scene objects such as Group and Scene to position and display our GUI items.

#### 4.3.9. javafx.scene.image

We will be using classes from javafx.scene.image such as Image and ImageView for the displaying of images and icons in our GUI.

#### 4.3.10. javafx.scene.input

This package will be used alongside javafx.event and java.util to handle user inputs from the mouse and keyboard.

#### 4.3.11. javafx.scene.media

We will be using this package's audio related objects to play sounds and background music.

#### 4.3.12. javafx.stage

In JavaFX, stages are the highest-level containers for GUI objects. Since we will be implementing our GUI in JavaFX, this package is fundamental.

### 4.4. Class Interfaces

#### 4.4.1. java.io.Serializable

As mentioned in 4.3, we will be using this interface to write and read certain objects to and from files using the ObjectOutputStream and ObjectInputStream classes.

#### 4.4.2. javafx.event.EventHandler

JavaFX's EventHandler interface allows developers to make handlers for different types of events by overriding its `handle(T event)` method. We will use this interface to handle clicks and keyboard presses.

#### 4.4.3. java.util.EventListener

While handling events, listeners will also be used to track mouse and keyboard events. We will be using EventListener's MouseListener and MouseMotionListener subinterfaces for mouse events, KeyListener subinterface for keyboard events and ActionListener for other events.

## 5. References

[1] Prechelt, Lutz. "Comparing Java vs. C/C++ efficiency differences to inter-personal differences". (2019). [pdf] pp.3-4. Available at: http://page.mi.fu-berlin.de/prechelt/Biblio/jccpp_cacm1999.pdf [Accessed 9 Nov. 2019].

[2] Oracle, "1 JDK 8 and JRE 8 Installation Start Here," *JDK 8 and JRE 8 Installation Start Here*, 06-Jan-2016. [Online]. Available:

https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html. [Accessed: 10-Nov-2019].

[3] Oracle, “Ücretsiz Java İndirme,” *Ücretsiz Java Yazılımını İndir*, 17-Apr-2018. [Online]. Available: https://www.java.com/tr/download/. [Accessed: 10-Nov-2019].

[4] B. Porter, J. van Zyl, and O. Lamy, “Welcome to Apache Maven,” *Maven*. [Online]. Available: https://maven.apache.org/. [Accessed: 10-Nov-2019].

[5] Oracle, *Java Platform Standard Edition 8 Documentation*. [Online]. Available: https://docs.oracle.com/javase/8/docs/. [Accessed: 10. Nov. 2019].

[6] Oracle, *JavaFX 8*. [Online]. Available: https://docs.oracle.com/javase/8/javafx/api/toc.htm. [Accessed: 10. Nov. 2019].