



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

| | |
|---|-----------|
| Introduction | 5 |
| Purpose of the system | 5 |
| Design Goals | 5 |
| Ease of Use and Learning | 5 |
| Runtime Efficiency | 5 |
| Portability | 5 |
| Modifiability | 5 |
| High-level Software Architecture | 6 |
| Subsystem Decomposition | 6 |
| Hardware/Software Mapping | 7 |
| Persistent Data Management | 7 |
| Access Control and Security | 8 |
| Boundary Conditions | 8 |
| Initialization | 8 |
| Termination | 8 |
| Errors | 9 |
| Subsystem Services | 9 |
| Presentation Layer | 9 |
| Business Layer | 9 |
| Data Layer | 10 |
| Low-level Design | 10 |
| Object Design Trade-offs | 10 |
| Ease of Use vs. Functionality | 10 |
| Memory vs. Performance | 10 |
| Modifiability vs. Robustness | 11 |
| Final Object Design | 11 |
| Presentation Layer | 13 |
| ScreenViewer | 13 |
| Screen | 13 |
| PopupScreen | 14 |
| GameMenuScreen | 14 |
| TradeScreen | 14 |
| EndGameScreen | 14 |
| GameScreen | 15 |
| MainMenu | 15 |
| HowToPlay | 16 |

| | |
|--|----|
| SettingsScreen | 16 |
| GameOptionsScreen | 16 |
| Business Layer | 17 |
| Game Controller Package | 18 |
| GameController | 18 |
| SettingsController | 18 |
| InputManager | 19 |
| SoundManager | 19 |
| Game Model Package | 19 |
| GameModel | 19 |
| GameObject | 20 |
| GameTile | 20 |
| Hexagon | 20 |
| Edge | 20 |
| Vertex | 21 |
| Building | 21 |
| Settlement - Road - City | 22 |
| Bank | 22 |
| DevCard | 22 |
| VictoryPoint, Monopoly, RoadBuilding, KnightCard, YearOfPlenty | 23 |
| Dice | 23 |
| Trade, DomesticTrade, TradeWithBank | 23 |
| PlayerList | 24 |
| Player | 24 |
| BotPlayer | 25 |
| Data Layer | 26 |
| GameData | 26 |
| SettingsData | 27 |
| PreviousGameData | 27 |
| DataManager | 28 |
| Packages | 28 |
| java.util | 28 |
| java.io | 28 |
| java.lang | 28 |
| javafx.animation | 28 |
| javafx.application | 29 |
| java.css | 29 |
| javafx.event | 29 |
| javafx.scene | 29 |

| | |
|---------------------------|-----------|
| javafx.scene.image | 29 |
| javafx.scene.input | 29 |
| javafx.scene.media | 29 |
| javafx.stage | 29 |
| Class Interfaces | 29 |
| java.io.Serializable | 29 |
| javafx.event.EventHandler | 29 |
| java.util.EventListener | 29 |
| References | 30 |

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 a piece of land and building constructs, and the first player to reach 10 points wins the game for the classic mode and 20 points wins the game for the crazy mode. 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

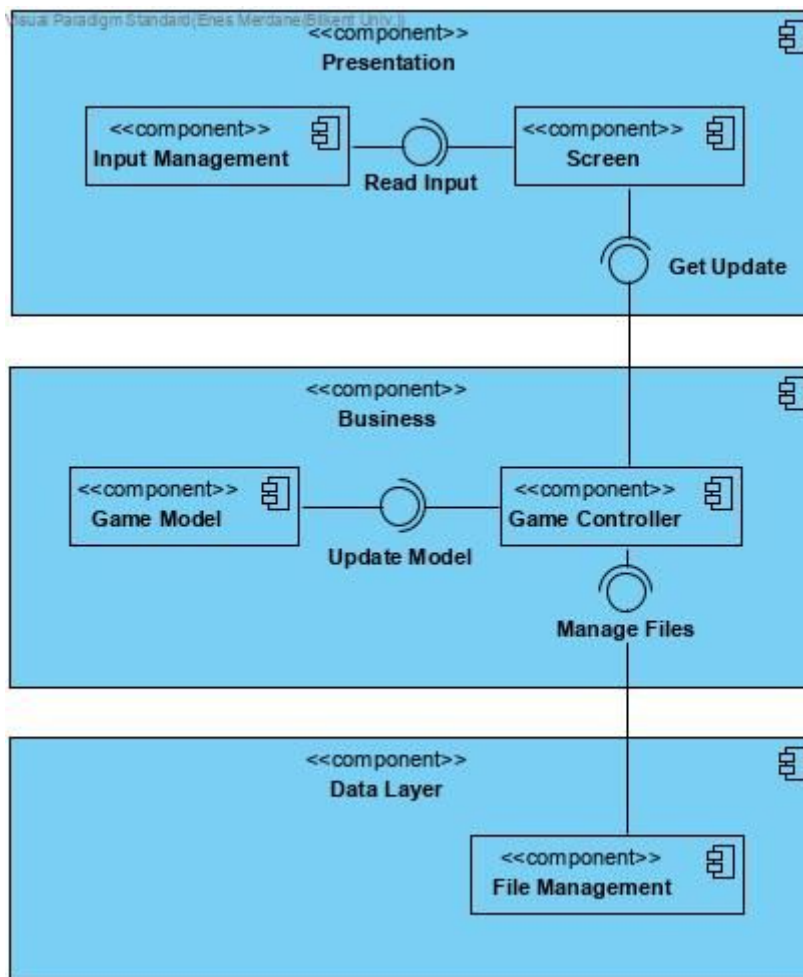


Figure 1: Three Layer Structure and Components

We have chosen to apply three layer 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 much as possible. Therefore, we divide our system into 3 different layers named Presentation, Business and Data. All these subsystems will have different functions and have interactions between each other. Presentation component will provide user interface and input handling operations. The interaction with the user will be this subsystem's responsibility. Business Component will include Game Model and Game Controller operations. Game Controller will receive the actions from presentation layer, control these actions with Game Model component and return the state of the changes to the Presentation Layer. Game Controller will also be the component which communicates with Data Layer. Our last subsystem Data Layer will manage the interaction between the file system and game for our savings which are settings preferences and the data of last player 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.

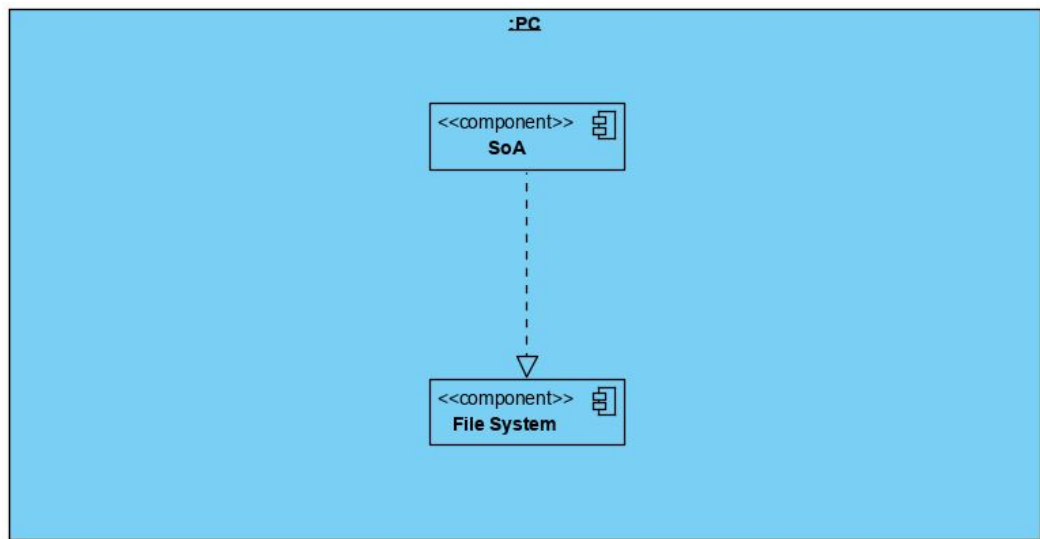


Figure 2: Deployment Diagram

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() pauseGame() exitGame() | trade() playCard() buyProgressCard() build() 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

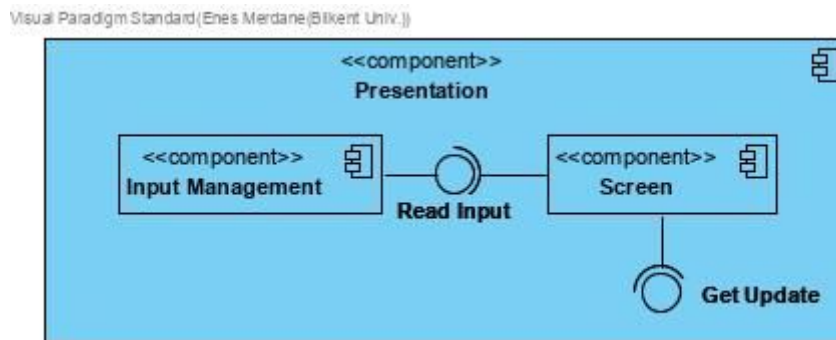


Figure 4: Presentation Layer

Presentation Layer represents the user interface and the user's interaction with this interfaces in Settlers of Anatolia.

When player clicks a component on the screen, Screen component's classes reads the input and calls the related methods from Game Controller component. According to the return of these methods, Screen component updates the view or not.

3.2. Business Layer

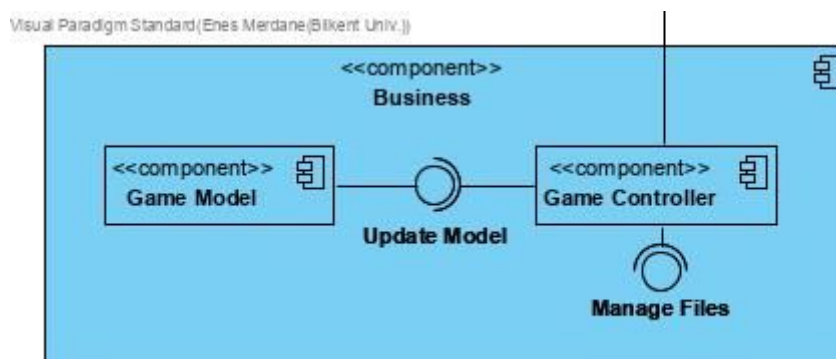


Figure 5: Business Layer - High Level Design

In Business Layer, there are 2 major components named Game Model and Game Controller. Game Controller is evoked when a method is called from Presentation Layer. Game Controller controls the states, determines the actions and returns these decisions to the Presentation Layer. Game Model component provides these states 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

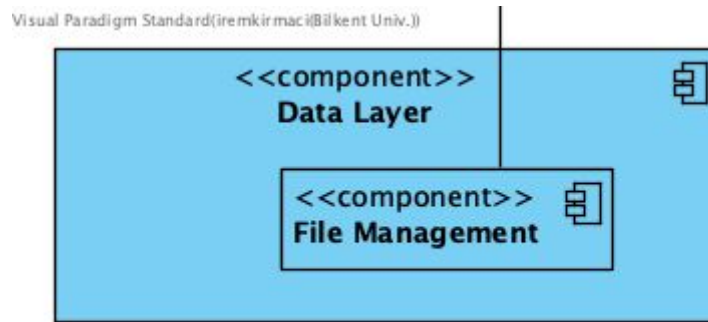


Figure 6: Data Layer - High Level Design

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

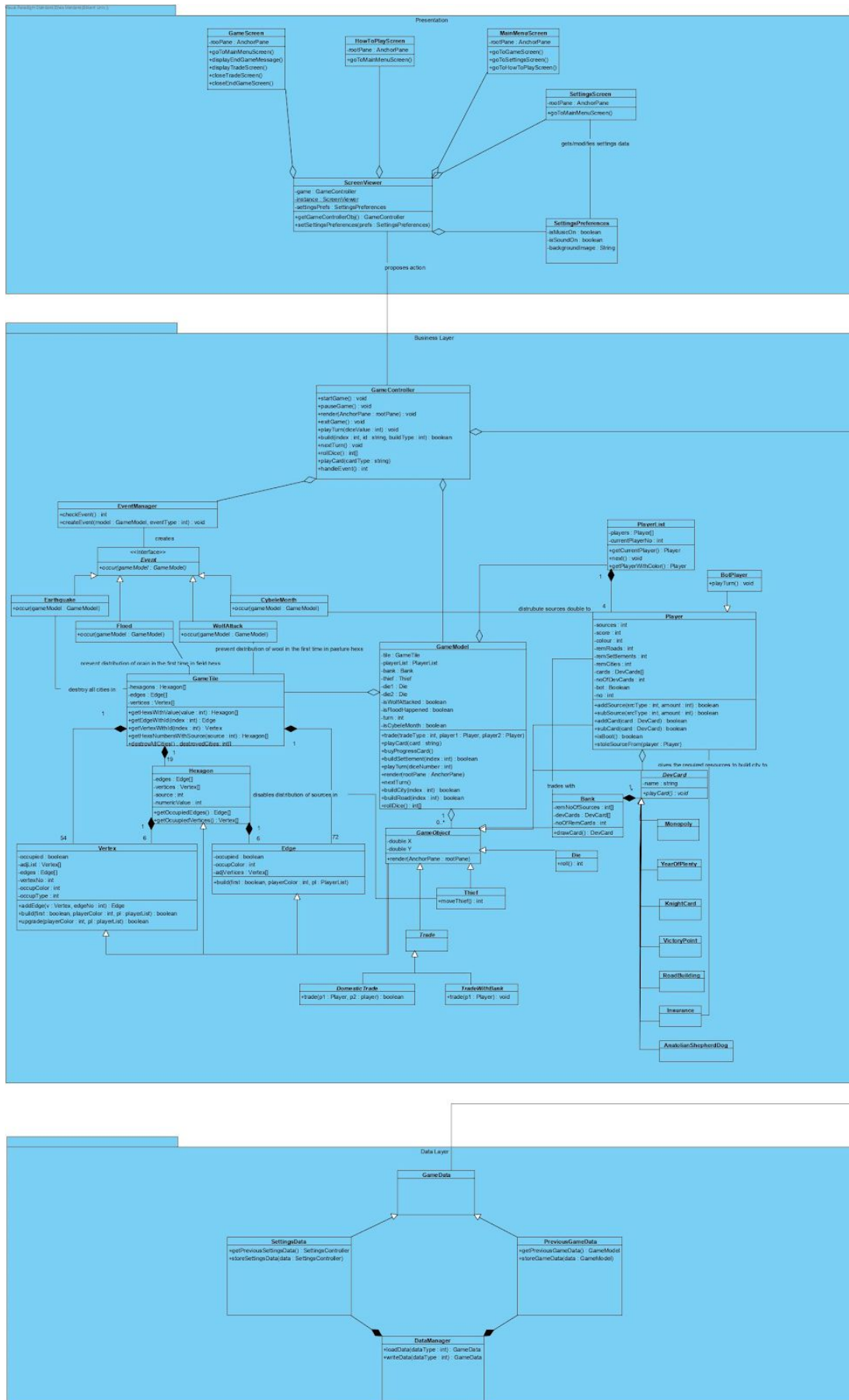
We believe that the game must be fast and efficient. Also, we do not want our players to lose interest waiting for the game to process their actions or for loading screens to finish. This is why we will use more RAM storage to keep references so that the instances have quicker access to other subsystems. 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 may want to add new features or to use newer technologies in our design. Therefore, we need to do these changes easily. It is important for our project to manage to cope with its errors as well. However, since we will be adding new changes, modifiability has the higher priority in this trade-off.

4.2. Final Object Design

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



4.2.1. Presentation Layer

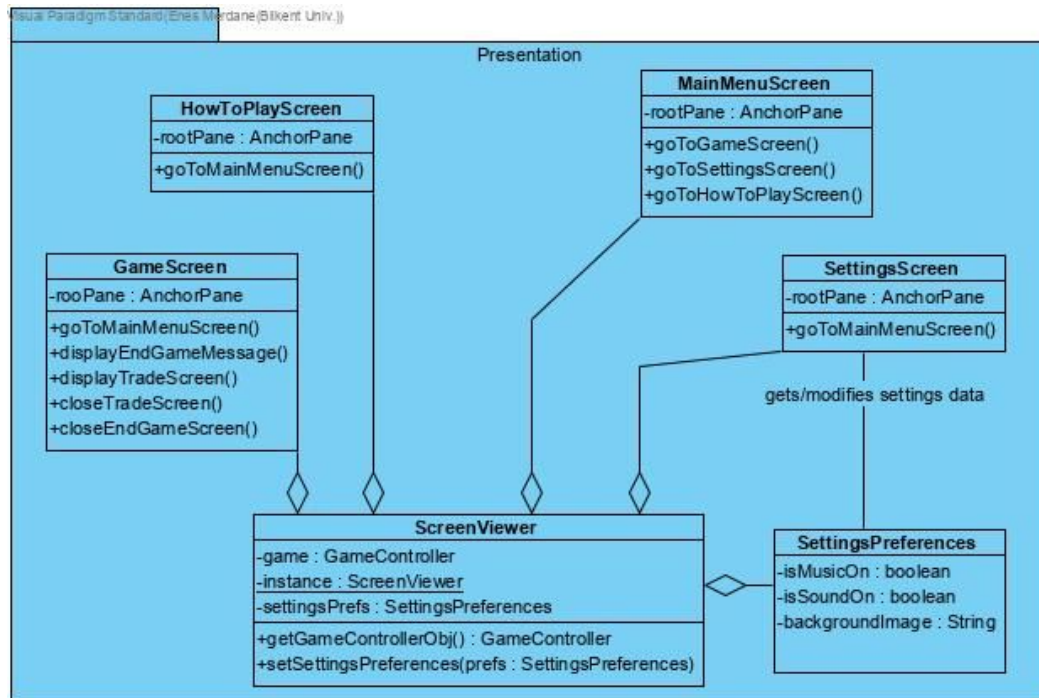


Figure 8: Presentation Layer Classes

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. **ScreenViewer** class initiates the **MainMenuScreen** and **Game Controller** objects. It assigns this **GameController** object to **game**. **ScreenViewer** class has **getGameControllerObj()** method which returns **game**. **setSettingPreferences(prefs: SettingsPreferences)** method is called by the **GameController** class to create **SettingsPreferences** object from the saved file coming from Data Layer. Finally, **ScreenViewer** instance is for other Screen Classes to access **ScreenViewer** and call its methods. In this class Facade Design Pattern is used. **GameController** class will have only one instance of this class.

4.2.1.2. GameScreen

Visual Paradigm Standard (Enes Merdane/Bilkent U)

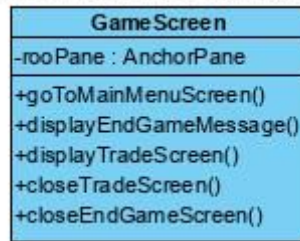


Figure 15: GameScreen Class

This class represents the main display of the game, which contains the elements of the board game, such as the board and cards. `rootPane` is the instance to access and draw new items to the current screen. Because we use fxml files to load layouts fast, we need `rootPane` to make modifications on fxml. `goToMainMenuScreen()` method initiates the `MainMenuScreen` class. `displayEndGameMessage()` draws an end game message to the current screen and disables all current buttons. `displayTradeScreen()` draws a trade screen to the current screen and disables all current buttons. `closeTradeScreen()` clears the trade screen from the screen and enables buttons again. `closeEndGameScreen()` clears the `endGameScreen` and initiates `MainMenu` screen.

4.2.1.3. MainMenuScreen

Visual Paradigm Standard (Enes Merdane/Bilkent U)

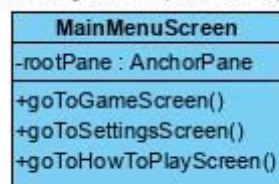


Figure 16: MainMenu Class

This class represents the Main Menu of the game, which opens when the game is initiated. This class has `rootPane` which is an instance of `AnchorPane` to be able to draw new items to the screen. This class has three main methods, which are `goToGameScreen()` which initiates the `GameScreen` class, `goToSettingsScreen()` which initiates `SettingsScreen` class and `goToHowToPlayScreen()` which initiates `SettingsScreen` class.

4.2.1.4. HowToPlayScreen

Visual Paradigm Standard (Enes Merdane/Bilkent U)

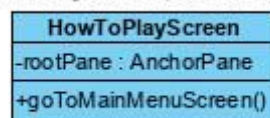


Figure 17: HowToPlay Class

This is the class for the screen that opens when the user clicks “How to Play” button while in the Main Menu. Other than the visual objects which will be displayed on the screen, this class has `rootPane`

object to manage screen operations as other Screen classes and `goToMainMenuScreen()` method which brings the main menu to the screen and initiates `MainMenuScreen` class.

4.2.1.5. SettingsScreen

Visual Paradigm Standard (Enes Merdane/Bilke)

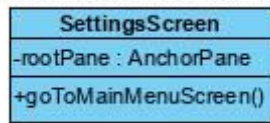


Figure 18: SettingsScreen Class

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 with the method `goToMainMenuScreen()`.

4.2.1.6. SettingsPreferences

Visual Paradigm Standard (Enes Merdane/Bilke)

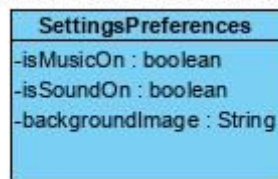


Figure 19: GameOptionsScreen Class

This class holds the settings preferences of the player. It is created with the creation of the `GameModel` class and is used by `SettingsScreen` to display current preferences. When a preference changes, the necessary actions are taken such as turning the game music/sounds on/off and changing the current `backgroundImage` preference.

4.2.2. Business Layer

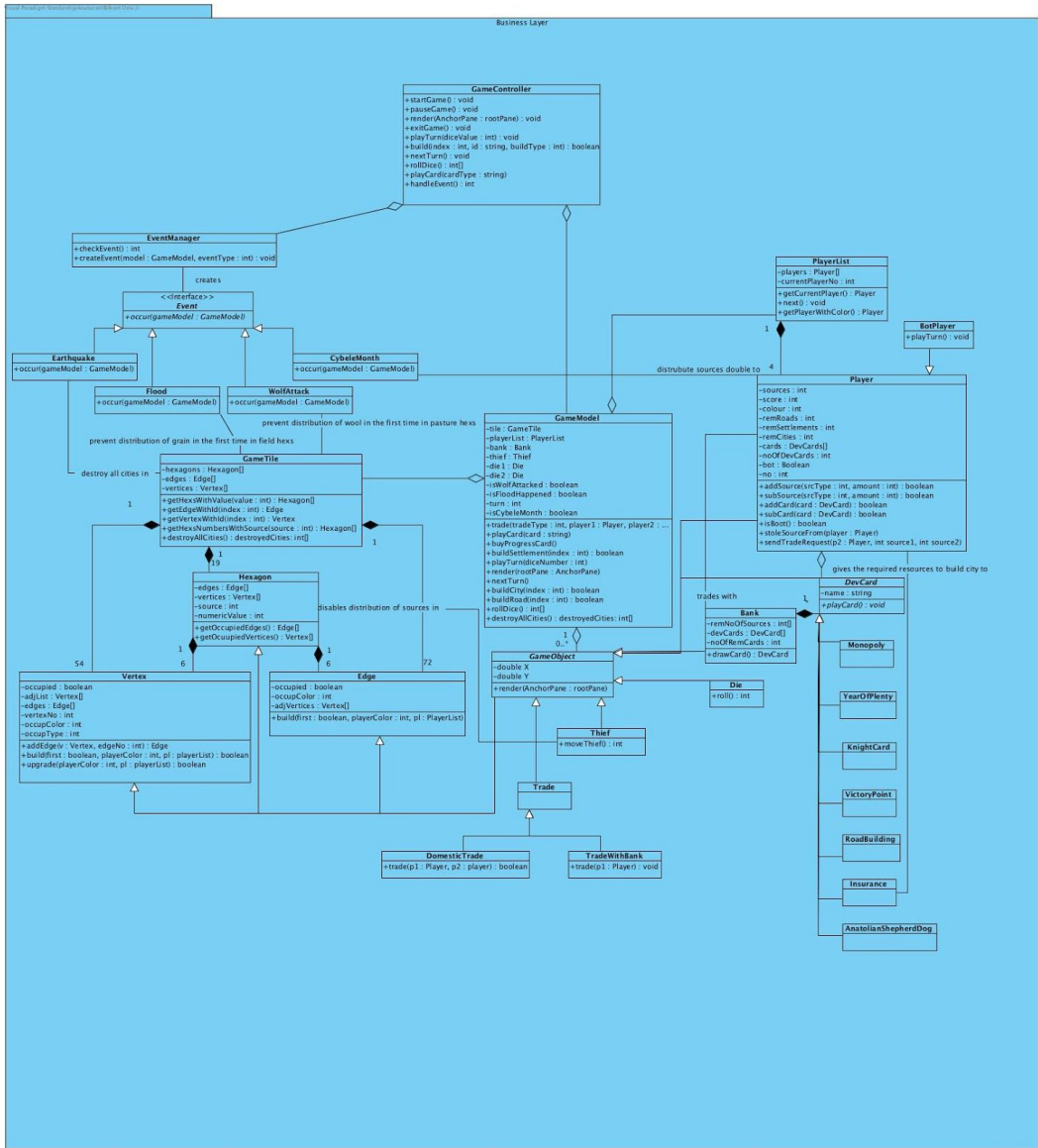


Figure 20: Business Layer Classes

4.2.2.1. Game Controller Package

4.2.2.1.1. GameController

Visual Paradigm Standard(goksuturan(Bilkent Univ.))

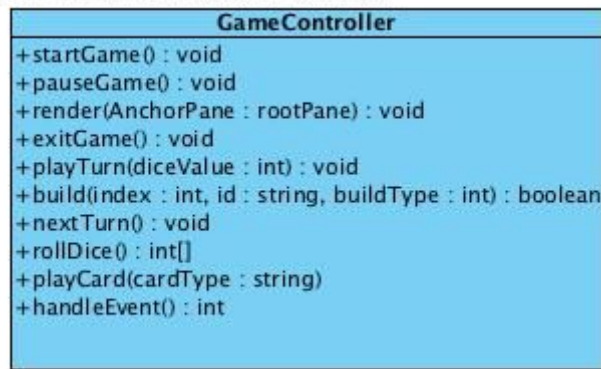
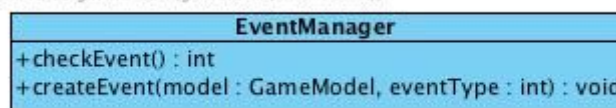


Figure 21: GameController Class

The GameController class, as its name suggests, controls the state of the game. It holds a GameModel instance to be able to control the game state and holds an EventManager instance which checks and handles the occurrence of the events consisting of disasters and CybeleMonth. It has 10 methods. `startGame()` starts the game by initializing the proper objects. `pauseGame()` pauses the game and saves it into a file. `render(AnchorPane: rootPane): void` is used to draw the objects of the game. `exitGame()` stops the game. `playTurn(diceValue: int) : void` first does the necessary actions depending on the value of the dice, and then lets the current player play their turn. `build(index : int, id: string, buildType : int) : boolean` is used to build a settlement, city or road. `index` specifies the index of the object that will receive the build action. `id` is a string that can either be “vertex” or “edge” that specifies the type of the object that will receive the build action. `buildType` is an integer that is used if `id` is “vertex” to specify whether or not the player is building a settlement or a city. `nextTurn()` passes the game to the next turn. `rollDice()` rolls the dice, and returns an array containing the values of each dice. `playCard(cardType : string)` is used to play a development card. `handleEvent() : int` handles the current event.

4.2.2.1.2. EventManager

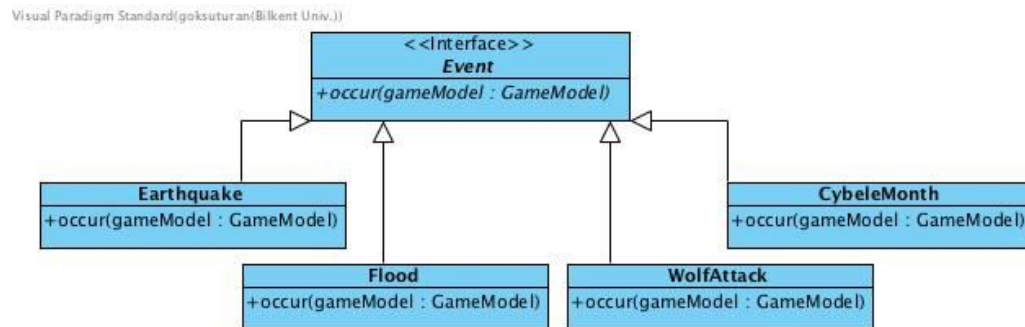
Visual Paradigm Standard(goksuturan(Bilkent Univ.))



EventManager class is responsible for managing the new events of the game. It checks with `checkEvent()` method, whether an event which can be of type Cybele or Disaster will occur. In this method a random integer in the range of 0 to 100 will be created. If this integer is in the

interval of 0 and 10, an event will occur and this method returns this integer. `createEvent()` method takes this integer as parameter. This method determines which event will be created and accordingly the corresponding method of the event class, will be called. If integer is in the range of 0 and 2, Earthquake will occur, in the range of 3 and 5, Flood will occur, in the range 6 and 8, WolfAttack will occur and in the range of 9 and 10 CybeleMonth will occur.

4.2.2.1.3. Event - CybeleMonth - Earthquake - Flood - WolfAttack



Event interface regulates the Earthquake, Flood, WolfAttack and CybeleMonth events. In there Abstract Factory design pattern is used. All of them has `occur(gameModel : GameModel)` method according to their own responsibilities. The occur method of Earthquake calls the `destroyAllCities()` method of `GameModel`. The occur method of Flood change the value of `isFlood` attribute which is in the `GameModel` to true. The occur method of WolfAttack change the value of `isWolfAttacked` which is in the `GameModel` to true. The occur method of CybeleMonth change the value of `isCybeleMonth` attribute which is declared in `GameModel` to true, which doubles the resource production gains. Event interface's occur method calls the corresponding occur method of Earthquake, Flood, WolfAttack and CybeleMonth.

4.2.2.2. Game Model Package

4.2.2.2.1. GameModel

Visual Paradigm Standard(göksuturan(Bilkent Univ.))



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 dice, to play or buy some cards and in order for a particular player to play its turn.

`trade(tradeType : int, player1 : Player, player2 : Player)` is used when the current player wants to trade. `tradeType` is an integer which signifies the type of trade. It will have value 0 if the trade is to be done with player or 1 if the trade is to be done with the bank. The following two variables correspond to the targets of the trade. `player1` is the player sending the trade, and `player2` is the one receiving the trade. If `tradeType` is 1, then the `player2` parameter will be null, meaning that the player is trading with the bank. `playCard(card: string)` plays the development card with the given name. `buyProgressCard()` is called when the current player wants to buy a development card. `buildSettlement(index : int) : boolean` is called when the current player wants to build a settlement. It receives the index of the vertex as parameter, and returns a boolean that signifies whether or not the build action was successful. `playTurn(diceNumber: int) : void` first does the necessary actions depending on the value of the dice, and then lets the current player play their turn. `render(rootPane : AnchorPane)` draws each GameModel that is to be drawn. `nextTurn()` passes the game to the next turn. `buildCity(index : int) : boolean` and `buildRoad(index : int) : boolean` are

similar to the buildSettlement method. `rollDice() : int[]` rolls the dice and returns the values in an integer array. Finally, `destroyAllCities() : int[]` destroys all cities in the map and returns the indices of the cities in an integer array.

4.2.2.2.2. GameObject

Visual Paradigm Standard(gol

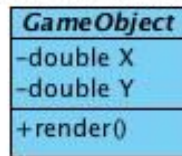


Figure 26: GameObject Class

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

Visual Paradigm Standard(goksuturan(Bilkent Univ.))



Figure 27: GameTile Class

GameTile is the class in which all the Edge, Vertex and Hexagon objects of the map are initialized and connected to each other. It has 19 hexagons, 72 edges and 54 vertices.

The GameTile class has 5 methods.

`getHexsWithValue(value : int): Hexagon[]` returns the Hexagon objects that have the given value as their number. `getEdgeWithId(index : int): Edge` returns the Edge object corresponding to the given index. `getVertexWithId(index: int): Vertex` returns the Vertex object corresponding to the given index. `getHexsNumbersWithSource(source: int): Hexagon[]` returns the Hexagon objects that produce the source given as the parameter. `destroyAllCities(): int[]` destroys all cities built on the map, and returns the indices of the destroyed cities as an array.

4.2.2.2.4. Hexagon

Visual Paradigm Standard(goksuturan(Bilkent Univ.))

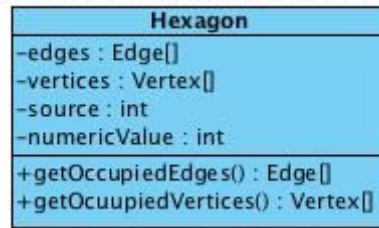


Figure 28: Hexagon Class

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. `getOccupiedEdges()` and `getOccupiedVertices()` methods return the edges and vertices in the hexagon that are occupied, respectively.

4.2.2.2.5. Edge

Visual Paradigm Standard(goksuturan(Bilkent Univ.))



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.

Beside its getter methods, edge has a `build(first : boolean, playerColor: int, pl: PlayerList): boolean` that is used to build a road on the edge. `first` parameter indicates if this method is called during the first turns. `playerColor` specifies the player that is trying to build the road, and `pl` is a `PlayerList` that the method uses to reach the current player and subtract their resources.

4.2.2.2.6. Vertex

Visual Paradigm Standard(goksuturan(Bilkent Univ.))



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.

The Vertex class has 3 methods, beside the getters. `addEdge(v : Vertex, edgeNo : int) : Edge` constructs an edge between the parameter vertex and the vertex that this method is called from, adds this new edge to both vertices' `edges` lists, and returns this edge. This method is used during the creation of the map. `build(first: boolean, playerColor: int, pl: PlayerList) : boolean` is called when a player tries to build a settlement on the vertex. the `first` parameter indicates if it is called during the first turns of the game, in which adjacency and resource conditions are not checked during build operations. `playerColor` indicates the player that is trying to build on the vertex. `pl` is the `PlayerList` that the method uses to reach the player that is trying to build the vertex, to subtract resources from them. `upgrade(playerColor: int, pl: PlayerList) : boolean` is similar to the build method, but it is used to upgrade a settlement to a city.

4.2.2.2.7. Bank

Visual Paradigm Standard(goksuturan/Bilker

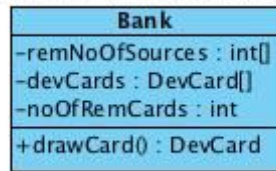


Figure 33: Bank Class

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.

4.2.2.2.8. DevCard

Visual Paradigm Standard(goksuturan/Bilker

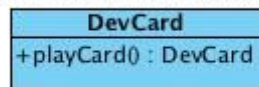


Figure 34: DevCard Class

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.

4.2.2.2.9. VictoryPoint, Monopoly, RoadBuilding, KnightCard, YearOfPlenty

Visual Paradigm Standard(goksu



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.

4.2.2.2.10. Dice

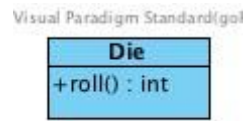


Figure 36: Die Class

Die class has a method called `roll()` to return result of dice. It is also a GameObject.

4.2.2.2.11. Trade, DomesticTrade, TradeWithBank

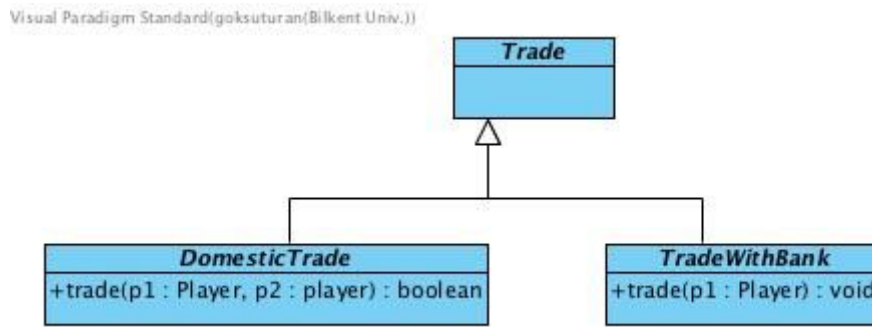


Figure 37: Trade Classes

The Trade class is an abstract class. It is generalized 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.

4.2.2.2.12. 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. By the `getPlayerWithColor` operation, the player with the color passed in as a parameter, is returned.

4.2.2.2.13. 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 held 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 steal 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.

4.2.2.2.14. BotPlayer

Visual Paradigm Standard(goksutur.



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

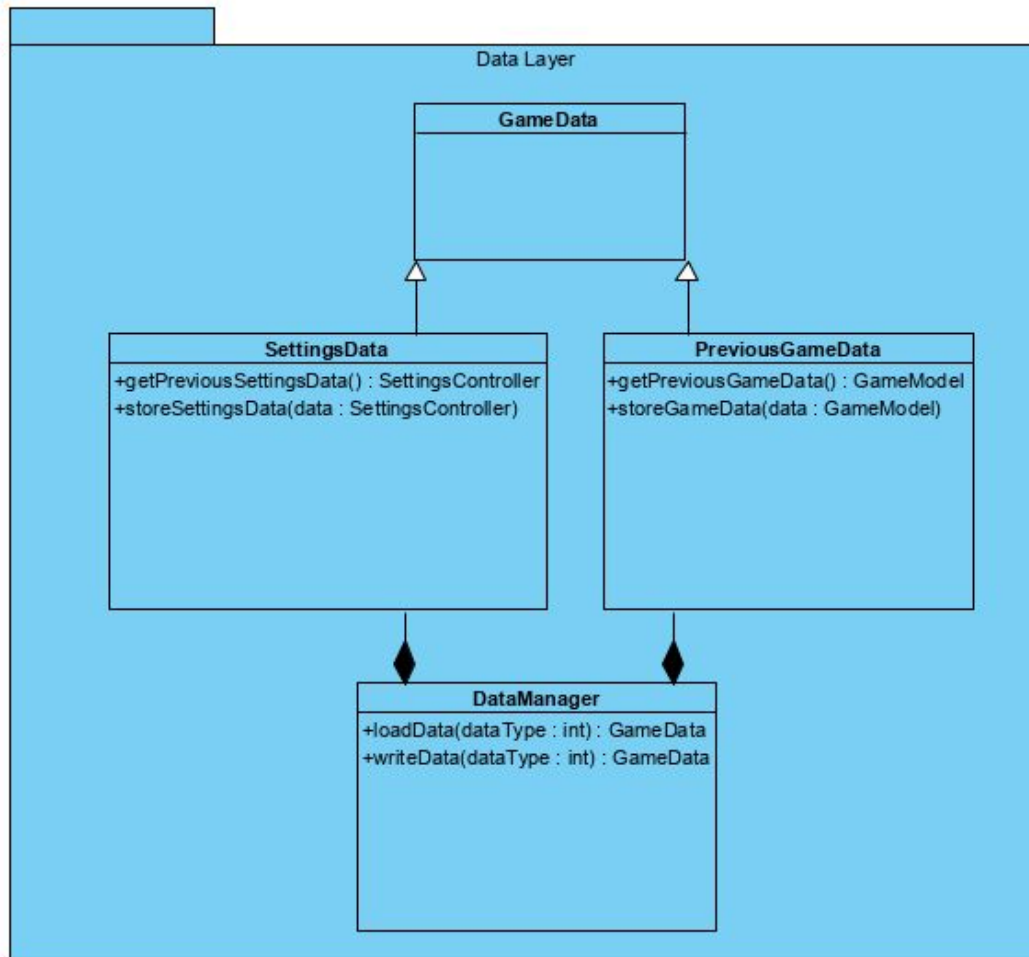


Figure 41: Data Layer Classes

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

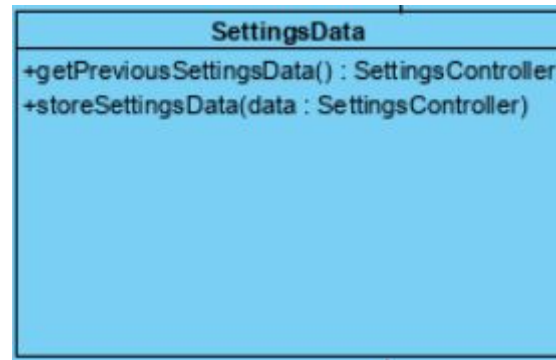


Figure 43: SettingsData Class

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 SettingsController instance and stores it into the file system.

4.2.3.3. PreviousGameData

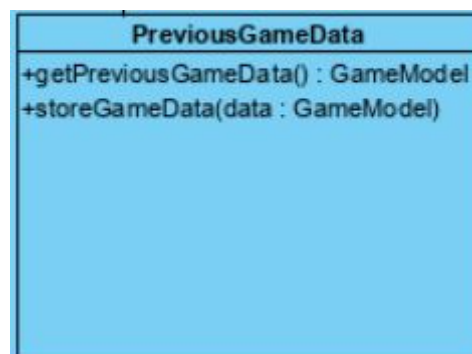


Figure 44: PreviousGameData Class

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

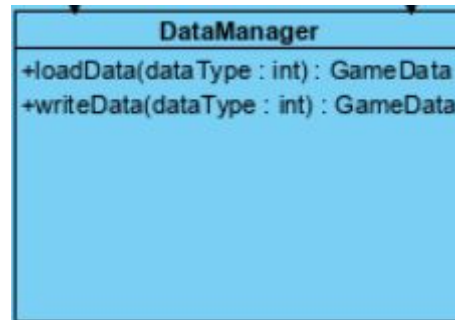


Figure 45: DataManager Class

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. Improvement Summary

In this iteration, we added new features to our project. This is why we had to add new classes and update the existing classes so that the software could correctly run our new features. Because of this, our low-level design has changed.

Furthermore, completing the implementation stage of the first iteration has provided us with a better idea of how our design should be. We have updated our low-level and high-level designs so that they are more compatible with our implementation.

In the previous iteration, within the Business Layer we had two separate packages called GameController and GameModel package. In this particular iteration, we put together these two packages into Business Layer.

In the presentation layer, now, we have input management component so we are sending the actions to Business Layer through Screen Management component to check the validity of these actions. If the action is valid we display the result of this action on the presentation layer.

6. 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].