Bilkent University

Department of Computer Engineering

CS319 Term Project

Group: 1I-TM

Spring 2020

Design Report

Team Members

Rafi Çoktalaş

Zeynep Cankara

Kamil Gök

Efe Macit

Arda Gültekin

Instructor: Eray Tüzün

Teaching Assistant(s): Alperen Çetin

Contents

Design Report

Terra Mystica

1. Introduction

1.1. Purpose of Terra Mystica

Terra Mystica is designed to accurately and completely digitalize the Terra Mystica board game. Terra Mystica will have all the components and game mechanics of his real world counterpart. Being in a digital environment adds flexibility and room for new features to be added.

Terra Mystica will be a multiplayer game and will be played online via Internet connection by two to five people. Terra Mystica will be compatible with Windows and MacOS.

The intended audience for Terra Mystica is only limited by knowing how to use a computer and viable Internet connection. Having a large scope will be kept in mind in external documentations.

1.2. Design Goals

1.2.1. Documentation

1.2.1.1. Internal Documentation

Each class and method will have javadoc comments which can later be used to generate an automatic documentation in HTML format [1]. Moreover, every method that contains an algorithm will have the algorithm in the source code as a pseudocode. This is to increase the readability of the code for the developers. However, this approach will also increase the writability as information in other classes and methods will be easy to understand for the developers all the time.

Internal documentation also includes pre-meeting documents in which the duration, topics and goal of a meeting is depicted and post-meeting documents which is the summary of a meeting. Post-meeting documents consist of the tools to be used and coding practices to follow.

1.2.1.2. External Documentation

Terra Mystica will have an in game help screen and also a tutorial document for users. Also the taken approaches, list of algorithms used will be documented for fellow developers.

1.2.2. Usability

The user interface will consist of 4 screens: login, main menu, help, and game. Other than the game screen, the screens will have minimal graphical elements to ease the entry of a user to the game. Also these screens will consist of buttons that are accessed directly and will equip users with a go back button thus providing users with simple navigation.

The game screen will inevitably have complex graphical elements to mimic the real board game. To overcome any inconvenience regarding usability several strategies will be employed. Firstly, a status label will always be visible on the game screen. The status label will narrate the last event,e.g., a message saying that the player is trying to take an action that is against the rules or simple messages explaining the change in the game board resulting from the last action. Secondly, the components will have distinct images and labels. Thirdly, the turn order and whose turn currently is being played will be shown all the time. Lastly, all components of the game which do not have to be visible at all times will be hidden to make room for those that have to be visible all the time. By doing so, players will have a cleaner graphical interface to see all the time. The hidden components will be accessible via distinct buttons representing them.

1.2.3. Robustness

The program will not act on invalid inputs on the game screen, i.e., a player trying to take action in another player’s turn. Furthermore, players will be notified why the action they have tried to take is invalid.

1.2.4. Extensibility

Even though Terra Mystica is a highly complex board game, there is always room for improvement; therefore the development will proceed in a way that will ease the addition of new functionalities or classes. So as to do so, fewer and cleaner dependencies; therefore, low cohesion and high coupling, will always be aimed for. Moreover, interfaces will be used to have further extensibility.

1.2.5. Performance

1.2.5.1. Response Time

In the game, players will see the outcome of the action they have taken after less than 1 second. Also other players will also be able to see the outcome of the action in a similar time. Moreover, the outcome of the action on the game will be narrated in the status bar. The status bar will be updated within 1 second when an action is happened or failed.

1.2.5.2. Memory

Terra Mystica will require less than 250 MB of space. To achieve this, algorithms will be investigated in terms of their space complexity and image files will be compressed.

1.2.5.3. Network

Users should be able to connect and play the game via a network connection with 1 Mbps bandwidth or higher.

1.2.6. Maintainability

The application will be easy to maintain. In order to achieve this, the source code will go under several reviews after it is written and before it is added to the application. The source code will be tested for maintainability measures with help of tools like Testwell CMTJava [2].

2. High-level Software Architecture

2.1. Subsystem Decomposition

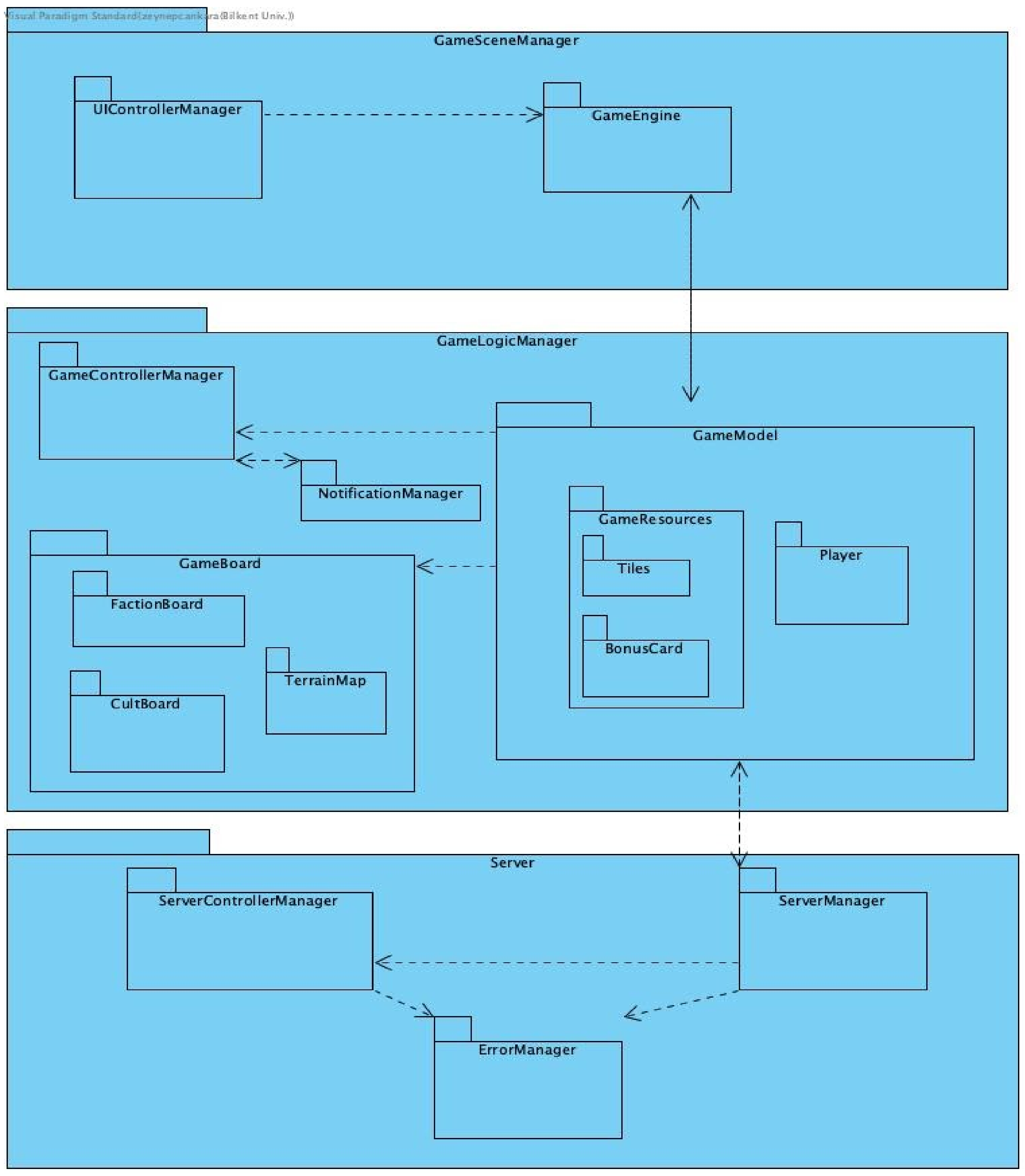


Figure 1: Subsystem Decomposition Diagram

We have decided to use 3-layer architectural style while designing our system. Our presentation layer consists of subsystems related with managing user interface components and game scenes hence named as *GameSceneManager,* which contains subsystems including *UIControllerManager* and *GameEngine*. Our application logic layer contains the application level game controllers and application domain entities hence named as *GameLogicManager*. The layer contains subsystems: *GameModel*, *GameControllerManager, GameBoard, NotificationManager*. Our server layer contains subsystems related with managing user and online play mode data hence named as *Server*. The layer contains subsystems *ServerManager, ServerControllerManager, ErrorManager*.

2.1.1 GameSceneManager

Subsystem responsible for managing the View of the game; includes subsystems responsible for handling the UI components and scenes.

2.1.2 UIControllerManager

Handles UI components via managing the controller logic of the game view. Contains controller classes belonging to the solution domain.

2.1.3 GameEngine

Subsystem responsible for establishing a connection in between game scenes and game logic.

2.1.4 GameLogicManager

Subsystem responsible for managing the game play logic. Contains subsystems related to the application domain.

2.1.5 GameModel

Subsystem responsible for creating an abstraction of the game entities.

2.1.6 GameControllerManager

Subsystem responsible for managing application level controllers; including the task of managing the game state.

2.1.7 NotificationManager

Subsystem managing messages related with the game play.

2.1.8 GameBoard

Subsystem responsible for holding the board state of the game, including the information related with terrain map, faction board and cult track.

2.1.9 Server

Our server communications will be handled by Java Admin SDK of Firebase Database. The server subsystem will contain subsystems managing client sessions and managing the state of the game by sending/receiving messages from the players. Additionally it will handle exception incidents.

2.1.10 ServerManager

Manages the interactions between game model and server.

2.1.11 ServerControllerManager

Contains controller classes related with processing the user responses in online play mode.

2.1.12 ErrorManager

Handles the incidents related to the server.

2.2. Hardware/Software Mapping

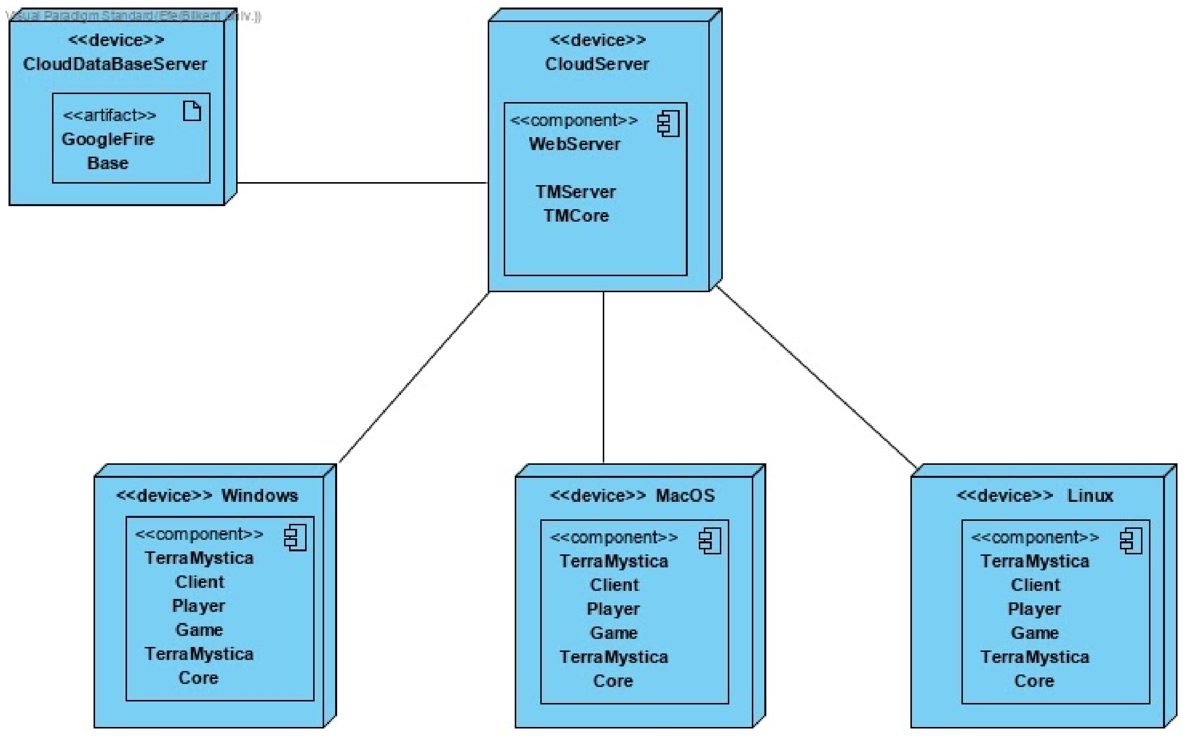


Figure 2: Hardware/Software Mapping Deployment Diagram

Above diagram is showing the hardware/software mapping. Game information is held in the FireBase Database and it is connected to Terra Mystica RestAPI server. Server holds the methods and structure of json files. Game is connected to servers via internet connection. Game can be played on 3 different OS platforms. It can be Windows, MacOs or Linux.

2.3. Persistent Data Management

Images will be kept in local hard disks of the users as the part of the client package. This will prevent unnecessary data flow. Initial map design is holded in a json file which is received from the server and game maps will be set up according to this initial file on the client. Database holds the registered players usernames and passwords. Server manages all the data which affects the multiplayer game environment. Server is independent from local applications. There are no saved or load game options in the game. Once the player has disconnected, the game cannot be loaded or if any errors occurred in the server, the game cannot be loaded in the players.

2.4. Access Control and Security

There is no single player mode, in multiplayer all players have to register and after that must login. There is no authorization in the game concept in the other words there is no administrator or player priority. All players have the same roles and there is no privilege between them. All players can take actions in just their turns, they are turned just a basic spectator on the player's turns.

2.5. Boundary Conditions

2.5.1. Initialization (Start-up)

In order to initialize the game, all UI componants such as fxml files, images and sound must load to the local system. The game will be in an executable program. When the user opens the executable of the game a fxml file associated with the main menu window called as mainMenu.fxml will be loaded. If a user wants to play a game in local mode localGame.fxml will be loaded which is the file associated with local play mode UI. After the images of the game loaded from the local hardware game will be ready to start reliably and stay stable throughout the game.

Multiplayer mode will require a different initialization process which involves verifying the user credentials to enable storing the user information in the cloud. The information stored in the cloud will involve previous gaming sessions which the users attended together with the players involved in the gaming session. Once the user clicks the “play online” button, the user will be directed to a login page where the user being asked to enter credential information which will be the user's gmail address with password. After user credentials are verified, the player will be directed to a new screen where the server matches the player with four random players. After four other players found it. The game will be initialized from the gameView.fxml file together with other game assets. During the initialization stage of the game server will be responsible for random initialization of the game board map and player resources. Afterwards each player’s instances will be controlled by the server in realtime.

2.5.2. Termination (Shutdown)

If the user plays the game in local play mode. The user either terminates execution of the game or presses the “Exit” button and returns to the main menu screen of the game. In case of multiplayer game mode, user termination might be the result of poor network connection. The user will encounter the main menu window when they exit the game due to network failure. Furthermore if the user wants to exit the game in the multiplayer mode, user presses the “Exit” button and returns to the main menu screen.

2.5.3. Exceptions

In this section we will address how we will react to the exceptions and failures which can occur independent of the game.

2.5.3.1. Loss of network connection

A user’s network can fail during any part of the online playing mode (login, player matching, online game). When a network connection failure happens the player will return to the main menu screen. If the connection failure happens during the online game all players will return to the main menu screen with a notification of which user disconnected from the game.

2.5.3.2. Failure of user authentication

Users will need to provide their authentication information for playing the game online from a server. The authentication method will be a combination of email-password combination since we are relying on Google Firebase Database for keeping user data in the cloud while getting Google’s promise of safety and security of the cloud database system. Authentication exceptions will occur if a user tries to provide inaccurate credential information which does not exist in the system database.

2.5.3.3. Inconsistent data

This type of system exception might occur when the player is in both online play and local play modes. Ideally one player must be notified by all the changes happening throughout the game. In case of local play mode the view might fail to update itself resulting displaying data inconsistent with the game state. In case of online play mode, the exception might occur if not all players modified by the update one player has on the game in real time.

2.5.3.4. Corrupted persistent data

This type of exception might occur when an fxml file fails to load UI assets including images and sounds. One of our most important design goals is Usability since we want out users to perform tasks safely, reliably and efficiently. Thus, when data corruption is noticed by the system the system stops trying to load the game and notifies the user about the corrupted data.

3. Subsystem Services

As mentioned in Subsystem Decomposition diagram, we used 3-Layer Architectural Style in our design. Main subsystems of the game are presentation layer, application logic layer and server layer subsystems. The layers responsible for the following subsystem services: Scene management, game flow and server management.

3.1. Scene Management Subsystem Service

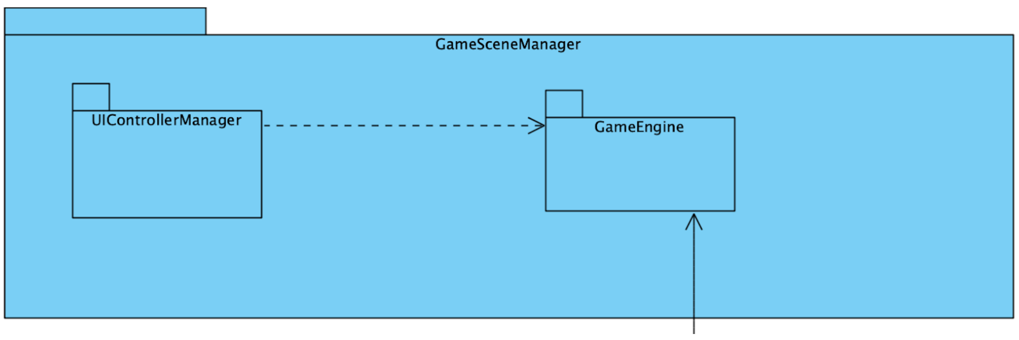


Figure 3: Scene Management Subsystem Service Focus

This layer contains subsystem services related to scene management service which includes managing UI components. UIControllerManager enables the scene transformations happening in between the scenes. The main scenes include Main Menu, Help Menu, Local

Game Play, Online Game Play. GameEngine subsystem works together with the application logic layer to update the view based on the game state.

3.2. Game Flow Subsystem Service

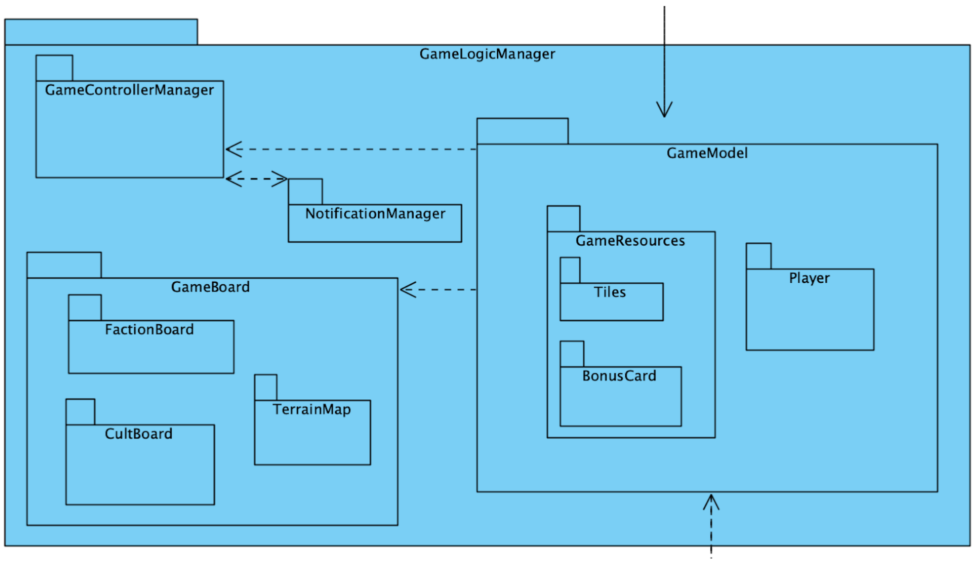


Figure 4: Game Flow Subsystem Service Focus

This subsystem service focuses on controlling the game logic and game states. The subsystem service consists of application domain entities such as terrain map, faction board, cult track, player and game resources as well as application domain controllers managing the game state transitions by handling messages between application domain entities.

3.3. Server Management Subsystem Service

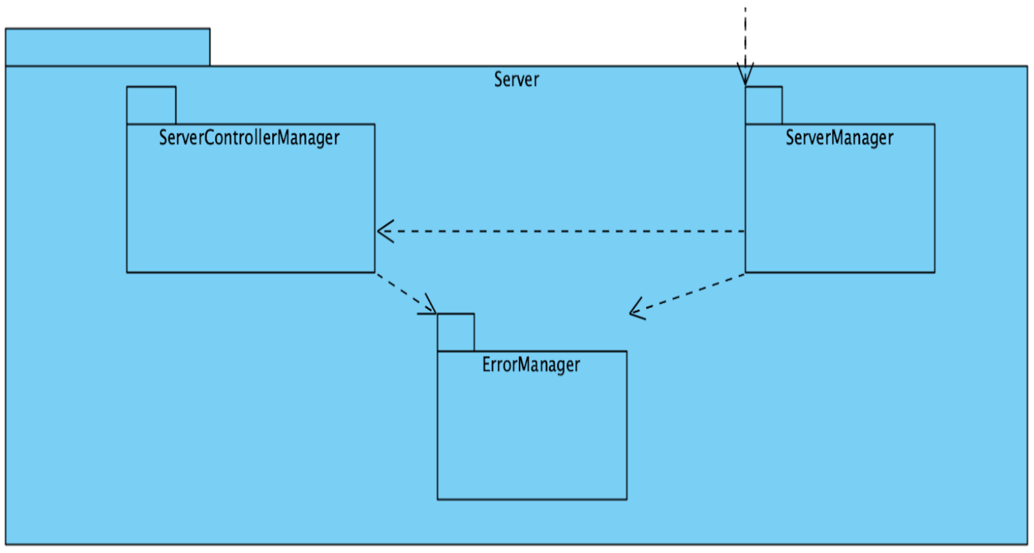


Figure 5: Server Management Subsystem Service Focus

Server Management Subsystem Service responsible for managing the game session data during the online play mode. The subsystem service is made possible by subsystems related with the Server layer of the game, including subsystems ServerManager, ServerControllerManager, ErrorManager. The ServerControllerManager processes the user responses while ServerManager handles the interactions between the game logic and server. ErrorManager provides maintenance.

4. Low-level Design

4.1. Object Design Trade-offs

4.1.1. Documentation vs. Development time

Detailed documentation is aimed in this project but the more detailed the documentation the more it requires time. Therefore, documentation may affect the development time negatively. However, detailed internal documentation will serve the project greatly in the long run in terms of maintenance and improvement; external documentation will introduce application to the users better.

4.1.2. Space complexity vs. Time complexity

In order to have the game require less space, the algorithms will prioritize space over time. The game consists of a relatively small number of inputs for its functions so time complexity can be compromised on. Furthermore, the data that will be used will be compressed before sending to the server to save space. Compression will take time but it will reduce the upload time and network usage.

4.1.3. Extensibility vs. Complexity

Having extensibility concerns may result in introduction of new interfaces and classes that would not be required otherwise. Thus, complexity of the source code may increase. This increase in complexity will not affect the application greatly yet being extensible may lead to great additions. Therefore, increase in extensibility will be favored more than a decrease in complexity.

4.1.4. Usability vs. Functionality

Other than the game screen, usability will be the greater concern since these screens will not house much functionality whereas on the game screen user interactions may become unpleasant to some extent in order to capture all the functionality of Terra Mystica. This will be minimized and heavily explained in the external documents and in the game help screen.

4.2. Class Diagram

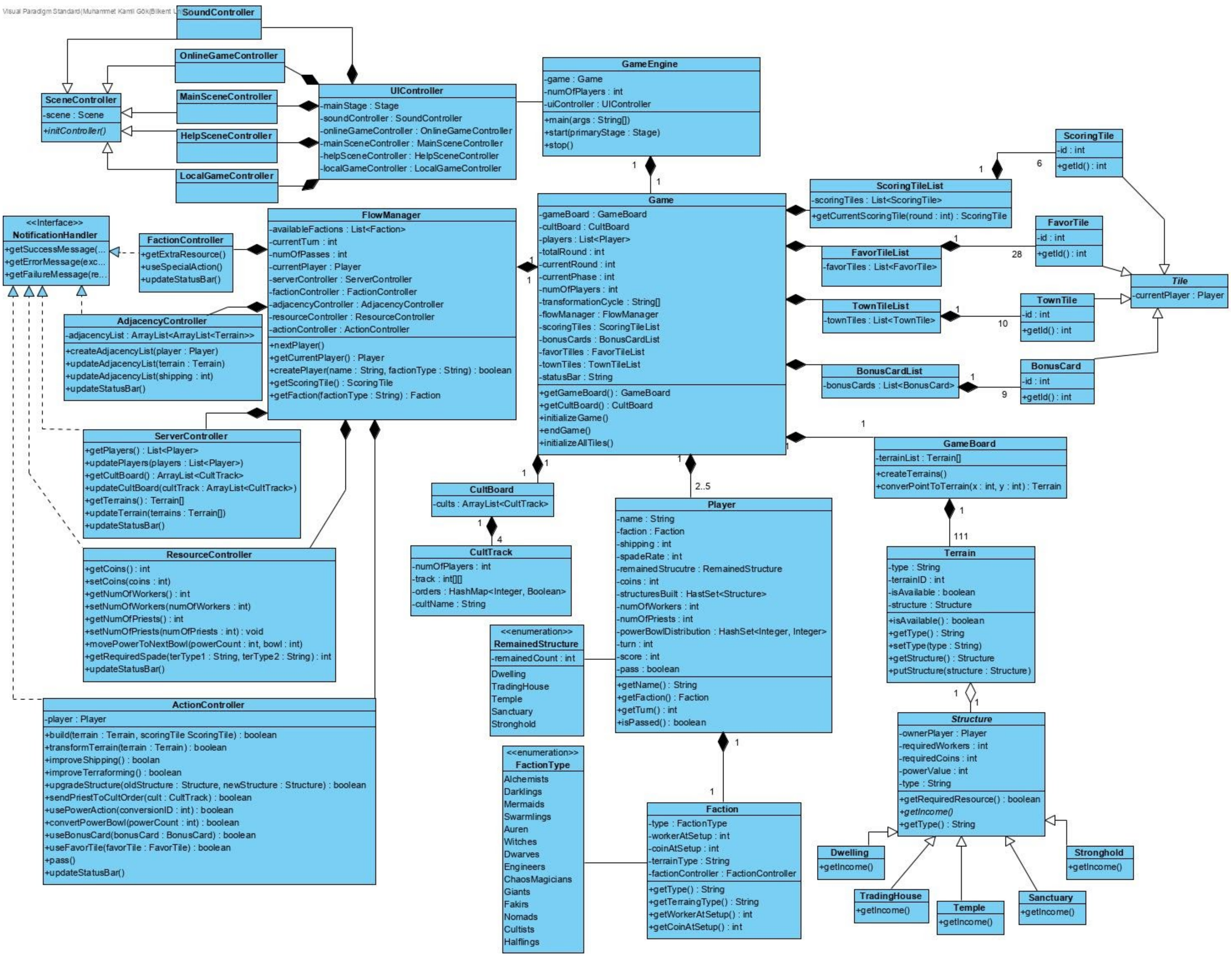

Visual Paradigm Standard(Muhammet Kamil Gök(Bilkent U
SoundController
OnlineGameController
SceneController
-scene : Scene
+initController()
MainSceneController
HelpSceneController
LocalGameController
UIController
-mainStage : Stage
-soundController : SoundController
-onlineGameController : OnlineGameController
-mainSceneController : MainSceneController
-helpSceneController : HelpSceneController
-localGameController : LocalGameController
GameEngine
-game : Game
-numOfPlayers : int
-uiController : UIController
+main(args : String[])
+start(primaryStage : Stage)
+stop()
<<Interface>>
NotificationHandler
+getSuccessMessage(...
+getErrorMessage(exc...
+getFailureMessage(re...
FactionController
+getExtraResource()
+useSpecialAction()
+updateStatusBar()
FlowManager
-availableFactions : List<Faction>
-currentTurn : int
-numOfPasses : int
-currentPlayer : Player
-serverController : ServerController
-factionController : FactionController
-adjacencyController : AdjacencyController
-resourceController : ResourceController
-actionController : ActionController
+nextPlayer()
+getCurrentPlayer() : Player
+createPlayer(name : String, factionType : String) : boolean
+getScoringTile() : ScoringTile
+getFaction(factionType : String) : Faction
AdjacencyController
-adjacencyList : ArrayList<ArrayList<Terrain>>
+createAdjacencyList(player : Player)
+updateAdjacencyList(terrain : Terrain)
+updateAdjacencyList(shipping : int)
+updateStatusBar()
ServerController
+getPlayers() : List<Player>
+updatePlayers(players : List<Player>)
+getCultBoard() : ArrayList<CultTrack>
+updateCultBoard(cultTrack : ArrayList<CultTrack>)
+getTerrains() : Terrain[]
+updateTerrain(terrains : Terrain[])
+updateStatusBar()
ResourceController
+getCoins() : int
+setCoins(coins : int)
+getNumOfWorkers() : int
+setNumOfWorkers(numOfWorkers : int)
+getNumOfPriests() : int
+setNumOfPriests(numOfPriests : int) : void
+movePowerToNextBowl(powerCount : int, bowl : int)
+getRequiredSpade(terType1 : String, terType2 : String) : int
+updateStatusBar()
ActionController
-player : Player
+build(terrain : Terrain, scoringTile ScoringTile) : boolean
+transformTerrain(terrain : Terrain) : boolean
+improveShipping() : boolean
+improveTerraforming() : boolean
+upgradeStructure(oldStructure : Structure, newStructure : Structure) : boolean
+sendPriestToCultOrder(cult : CultTrack) : boolean
+usePowerAction(conversionID : int) : boolean
+convertPowerBowl(powerCount : int) : boolean
+useBonusCard(bonusCard : BonusCard) : boolean
+useFavorTile(favorTile : FavorTile) : boolean
+pass()
+updateStatusBar()
Game
-gameBoard : GameBoard
-cultBoard : CultBoard
-players : List<Player>
-totalRound : int
-currentRound : int
-currentPhase : int
-numOfPlayers : int
-transformationCycle : String[]
-flowManager : FlowManager
-scoringTiles : ScoringTileList
-bonusCards : BonusCardList
-favorTilles : FavorTileList
-townTiles : TownTileList
-statusBar : String
+getGameBoard() : GameBoard
+getCultBoard() : CultBoard
+initializeGame()
+endGame()
+initializeAllTiles()
ScoringTileList
-scoringTiles : List<ScoringTile>
+getCurrentScoringTile(round : int) : ScoringTile
ScoringTile
-id : int
+getId() : int
FavorTileList
-favorTiles : List<FavorTile>
FavorTile
-id : int
+getId() : int
TownTileList
-townTiles : List<TownTile>
TownTile
-id : int
+getId() : int
BonusCardList
-bonusCards : List<BonusCard>
BonusCard
-id : int
+getId() : int
Tile
-currentPlayer : Player
GameBoard
-terrainList : Terrain[]
+createTerrains()
+converPointToTerrain(x : int, y : int) : Terrain
Terrain
-type : String
-terrainID : int
-isAvailable : boolean
-structure : Structure
+isAvailable() : boolean
+getType() : String
+setType(type : String)
+getStructure() : Structure
+putStructure(structure : Structure)
Structure
-ownerPlayer : Player
-requiredWorkers : int
-requiredCoins : int
-powerValue : int
-type : String
+getRequiredResource() : boolean
+getIncome()
+getType() : String
Dwelling
+getIncome()
TradingHouse
+getIncome()
Temple
+getIncome()
Sanctuary
+getIncome()
Stronghold
+getIncome()
CultBoard
-cults : ArrayList<CultTrack>
CultTrack
-numOfPlayers : int
-track : int[][]
-orders : HashMap<Integer, Boolean>
-cultName : String
<<enumeration>>
RemainedStructure
-remainedCount : int
Dwelling
TradingHouse
Temple
Sanctuary
Stronghold
<<enumeration>>
FactionType
Alchemists
Darklings
Mermaids
Swarmlings
Auren
Witches
Dwarves
Engineers
ChaosMagicians
Giants
Fakirs
Nomads
Cultists
Halflings
Player
-name : String
-faction : Faction
-shipping : int
-spadeRate : int
-remainedStrucutre : RemainedStructure
-coins : int
-structuresBuilt : HastSet<Structure>
-numOfWorkers : int
-numOfPriests : int
-powerBowlDistribution : HashSet<Integer, Integer>
-turn : int
-score : int
-pass : boolean
+getName() : String
+getFaction() : Faction
+getTurn() : int
+isPassed() : boolean
Faction
-type : FactionType
-workerAtSetup : int
-coinAtSetup : int
-terrainType : String
-factionController : FactionController
+getType() : String
+getTerraingType() : String
+getWorkerAtSetup() : int
+getCoinAtSetup() : int
1
1
1
6
1
28
1
10
1
9
1
1
1
1
2..5
1
4
1
111
1
1
1
1

Figure 6: Class Diagram(see Appendix)

4.2.1 GameEngine Class

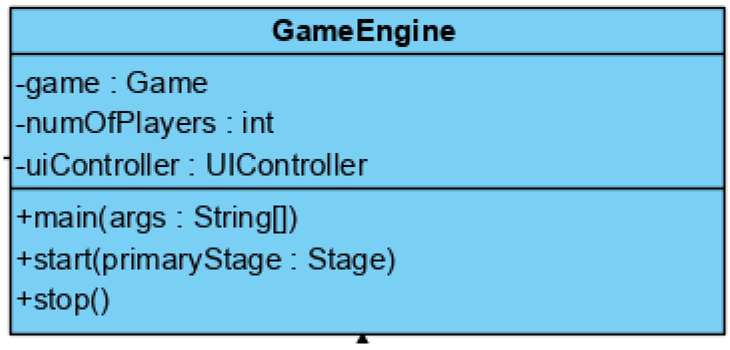


Figure 7: MainController Class Diagram

This class provides connection between the user interface package and object model. The instance of Game class is created in GameEngine class. Object model is reached through the instance of Game class.

4.2.1.1 Attributes of GameEngine Class

private Game *game*: It is used to keep the game object.

private static int numOfPlayers: It is used to store the number of players in the game.

private UIController uiController: It is used to connect UI with the object model.

4.2.1.2 Methods of GameEngine Class

public static void main(String args[]) : Starts the whole game.

public void start(Stage primaryStage) : Displays the primary stage of the game.

public void stop() : Stops the whole game and exits.

4.2.2. Game Class

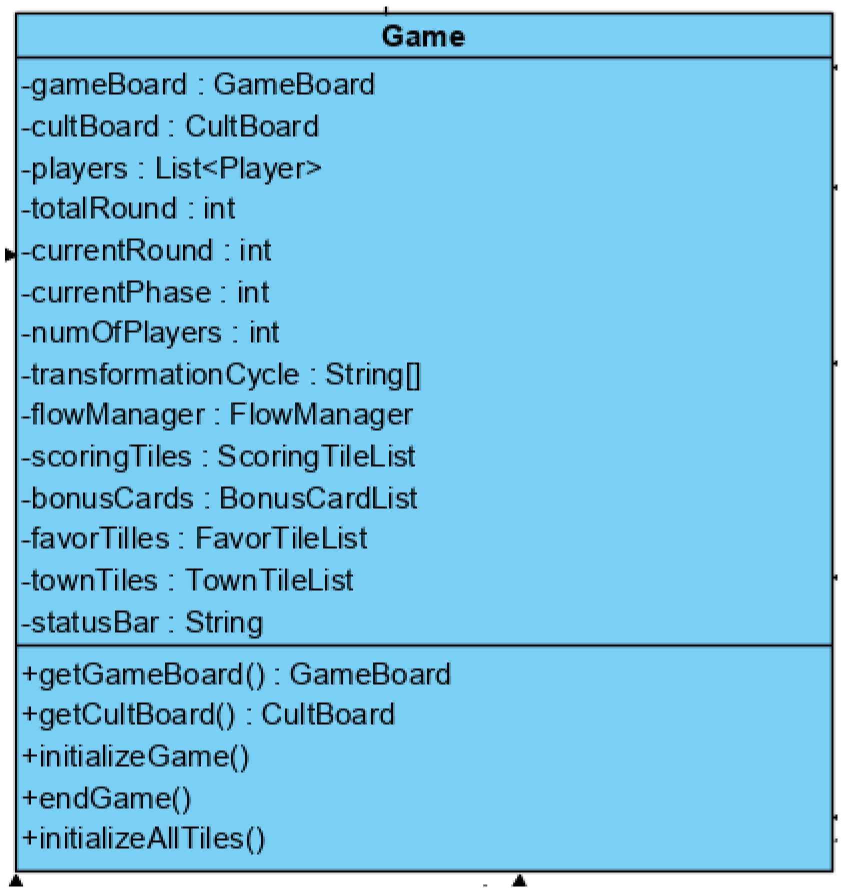

Game
-gameBoard : GameBoard
-cultBoard : CultBoard
-players : List<Player>
-totalRound : int
-currentRound : int
-currentPhase : int
-numOfPlayers : int
-transformationCycle : String[]
-flowManager : FlowManager
-scoringTiles : ScoringTileList
-bonusCards : BonusCardList
-favorTilles : FavorTileList
-townTiles : TownTileList
-statusBar : String
+getGameBoard() : GameBoard
+getCultBoard() : CultBoard
+initializeGame()
+endGame()
+initializeAllTiles()

Figure 8: Game Class Diagram

Everything related to the game is connected to the Game Class, it keeps track of the game progress with its attributes and functions.

4.2.2.1 Attributes of Game Class

private GameBoard gameBoard: It is used to store the changes in game board / game.

private CultBoard cultBoard : It is used to store the cult board progress.

private ArrayList<Player> players: It is used to store all the players playing the game.

private int totalRound : It is used to store the total rounds in the game.

private int currentRound : It is used to store current round number.

private int currentPhase : It is used to store current phase number.

private int numOfPlayers: It is used to store the total number of players playing the game.
private static String[] transformationCycle: It is used to keep the conversion cycle of different terrain types.
private FlowManager flowManager : It is used to store flow manager to control the flow of the game.
private ScoringTileList scoringTiles: It is used to store the scoring tile list.
private BonusCardList bonusCards: It is used to store the bonus card list.
private FavorTileList favorTiles: It is used to store the favor tile list.
private TownTileList townTiles: It is used to store the town tile list.
private String statusBar : It stores the status of the last action to send as notification message to the user.

4.2.2.1 Methods of Game Class

public GameBoard getGameBoard(): It is used to get the Game Board object.
public CultBoard getCultBoard(): It is used to get the Cult Board object.
public void initializeGame() : It is used to start the game after specifications are given by the users.
public void endGame(): It is used to end the game to direct the result screen.
public void initializeAllTiles(): It is used to initialize all default tiles in the game, specifically FavorTiles, ScoringTiles, BonusCards, TownTiles.

4.2.3. Player Class

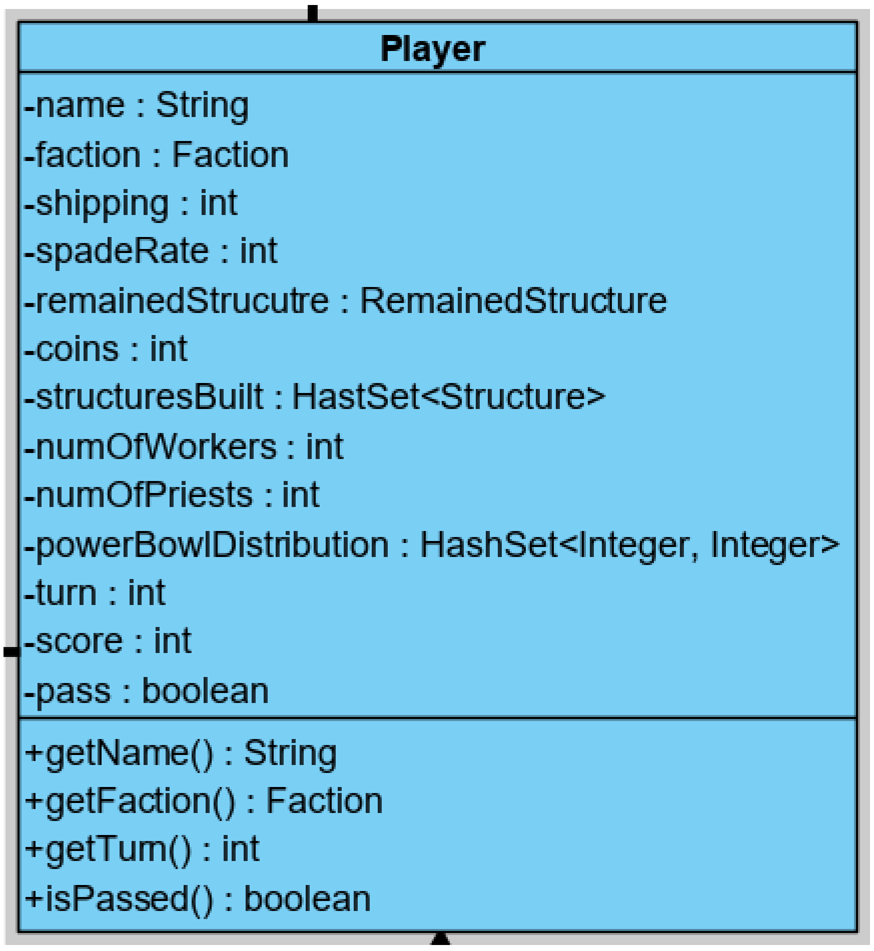


Figure 9: Player Class Diagram

Player class includes all attributes that belongs to a players and includes functions for the player actions. Also, the class has getter and setter functions for the attributes. This class’ functionalities constitute the main flow and actions of the game.

4.2.3.1 Attributes of Player Class

private String name: It is used to store the name of the player.

private Faction faction: It is used to store the chosen faction of the player.

private int shipping : It is used to store the shipping value of the player
private int turn: It is used to indicate the turn number of the player.
private int spadeRate : It is used to store the spade rate of the player.
private RemainedStructure remainedStructure : It is used to represent the remaining structures of the player.
private HashSet<Structure> structuresBuilt: It is used to store the structures built on the terrains.
private int score: It is used to store the score -victory points-.
private int coins: It is used to store the total number of coins owned.
private int numOfPriests: It is used to store the total number of priests owned.
private int numOfWorkers: It is used to store the total number of workers owned.
private HashSet<Integer,Integer> powerBowlDistribution: It is used to keep the number of power tokens in each of 3 power bowls.
private HashSet<Structure>structuresBuilt: It is used to keep track of what player built.
private boolean pass: It is used to indicate whether the player passed or not.

4.2.3.2 Constructor of Player Class

public Player(String name, Faction faction, int turn): Initializes player object with the given name, chosen faction and given turn number.

4.2.3.3 Methods of Player Class

public String getName(): Gets the name of the player.
public Faction getFaction(): Gets the faction of the player.
public int getTurn(): Returns the turn number of the player.
public boolean isPassed(): Checks if the player passes or not.

4.2.4 RemainedStructure Enum Class

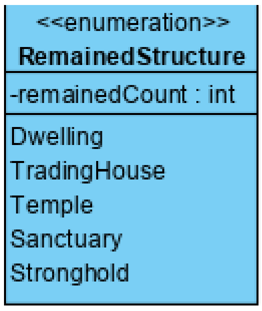


Figure 10: RemainedStructure Enum Class Diagram

This enumeration class is used to store numbers of available structures of different types which are Dwelling, TradingHouse, Temple, Sanctuary, Stronghold.

4.2.5 CultBoard Class

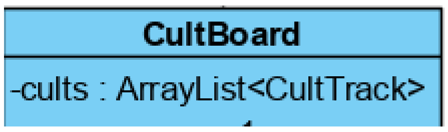


Figure 11: CultBoard Class Diagram

This class is used to represent the cult board status in the game. It has a single attribute which is list of CultTrack. Which will provide four different cult tracks in our game.

4.2.5.1. Attributes Of CultBoard Class

private <CultTrack> cults : The CultBoard class has a single attribute which stores 4 different cult tracks. And it is all stored in this array list.

4.2.6 CultTrack Class

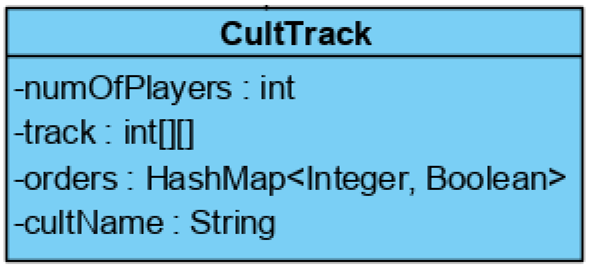


Figure 12: CultTrack Class Diagram

This class is used to keep track of a single cult track in the cult board. It stores the current situation of the individual cult track.

4.2.6.1 Attributes of CultTrack Class

private int numOfPlayers : It is used to store the number of players.

private int[][] track : It is used to store cult's track status.

private HashMap <Integer,Boolean> orders :

private String cultName : It is used to store the name of the cult.

4.2.7 Faction Class

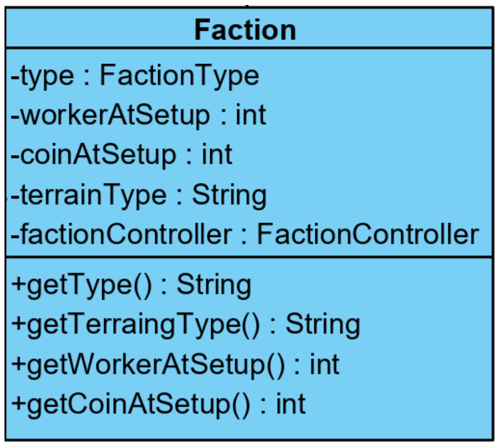


Figure 13: Faction Class Diagram

It is used to represent the Faction types in our game.

4.2.7.1 Attributes of Faction Class

private FactionType type : It stores the factions type.

private String terrainType: It stores the factions corresponding terrain type in string.

private int workerAtSetup: It stores the factions number of workers initially.

private int coinAtSetup: It stores the factions number of coins initially.

private FactionController factionController :

4.2.7.2 Methods of Faction Class

public String getType(): It returns the type of the faction in string.

public String getTerrainType(): It returns the terrain type in string.

public int getWorkerAtSetup(): It returns the number of workers in setup.

public int getCoinAtSetup(): It returns the number of coins at setup.

4.2.8 FactionType Enum Class

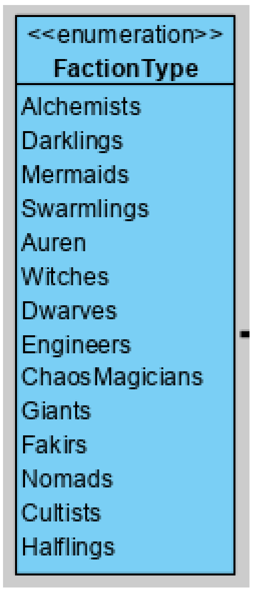


Figure 14: FactionType Enum Class Diagram

This enumeration class is used to store different types of factions which are necessary for faction-depended actions and resources.

4.2.9 GameBoard Class

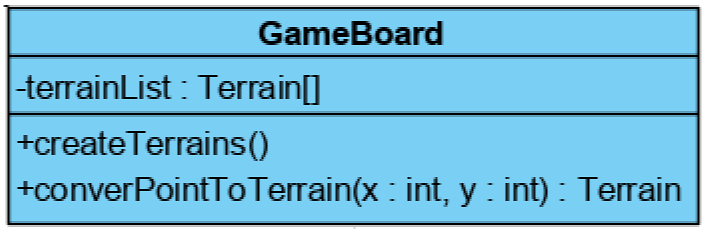


Figure 15: GameBoard Class Diagram

GameBoard class is used in our implementation to create the tiles in our game. It is also used to transform the terrains while playing the game.

4.2.9.1 Attributes of GameBoard Class

private Terrain[] terrainList : It is used to store all the terrains in the actual game board.

4.2.9.2 Methods of GameBoard Class

private void crateTerrains(): It creates all the terrains in the game and initialize to the terrainList.

private Terrain convertPointToTerrain(int x,int y): When player invokes transform method it comes here and changes the terrain.

4.2.10 Terrain Class

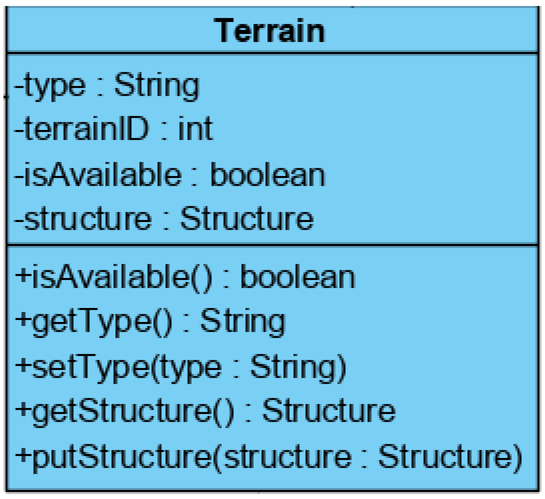


Figure 16: Terrain Class Diagram

This class is essence of the game since buildings done on the gameboard is a concern of the Terrain class.

4.2.10.1 Attributes of Terrain Class

private String type : It is used to represent the type of the terrain.

private int terrainID: Every different type terrain has different terrain ID. It is used to keep track of it.

private boolean isAvailable: It is used to check if the terrain is available.

private Structure structure : It is used to see the Structure which built on the terrain.

4.2.10.2 Methods of Terrain Class

public boolean isAvailable(): It returns the isAvailable variable.

public String getType(): It returns the type variable.

public void setType(String type): It changes the type variable of the class to the desired type.

public Structure getStructure(): It returns the structure variable.

public void putStructure(Structure structure): If the terrain is empty , or the building is being upgraded. This method is called to change the structure or put it first time.

4.2.11 Structure Abstract Class

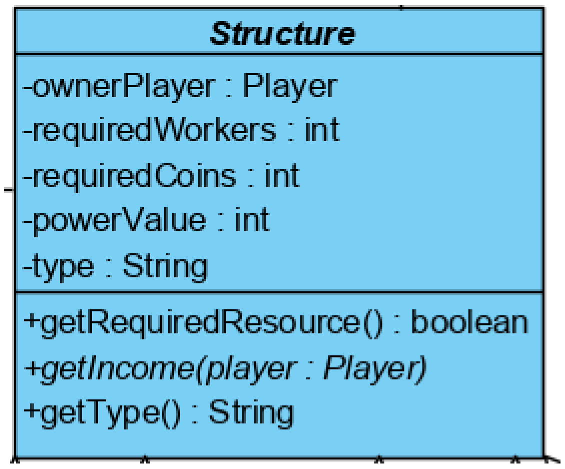


Figure 17: Structure Class Diagram

The structure class represents all the buildings in the game.

4.2.11.1 Attributes of Structure Abstract Class

private Player ownedPlayer: This variable shows who owns the Structure.

private int requiredWorkers : This variable shows the required worker number to be built.

private int requiredCoins: This variable shows the required coin number to be built.

private int powerValue : This variable represents the power value of the structure.

private String type: Since,there are different types of structures it represents which one is selected.

4.2.11.2 Method of Structure Abstract Class

public boolean getRequiredResource(): It checks whether the player can build the structure or not and gets the resource from the player if player can build.

public void getIncome(Player player): It adds the structures income to players income.

public void getType(): It returns the type of the structure.

4.2.12 Dwelling Class

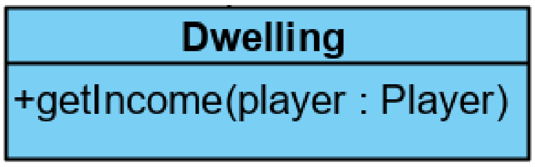


Figure 18: Dwelling Class Diagram

It is the child class of the structure and it's one of the structures in the game.

4.2.12.1 Methods of Dwelling Class

public getIncome(Player player) : It adds income to the players income where every structure have different income number.

4.2.13 Temple Class

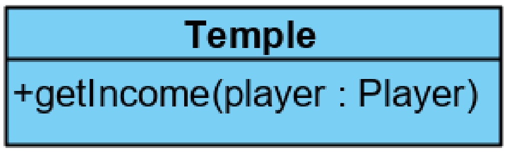


Figure 19: Temple Class Diagram

It is the child class of the structure and it's one of the structures in the game.

4.2.13.1 Methods of Temple Class

public getIncome(Player player) :It adds income to the players income where every structure have different income number.

4.2.14 TradingHouse Class

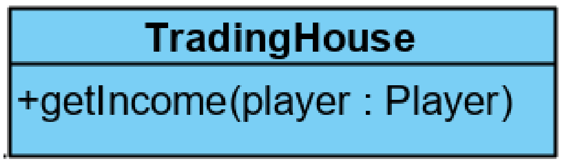


Figure 20: TradingHouse Class Diagram

It is the child class of the structure and it's one of the structures in the game.

4.2.14.1 Methods of TradingHouse Class

public getIncome(Player player) :It adds income to the players income where every structure have different income number.

4.2.15 Sanctuary Class

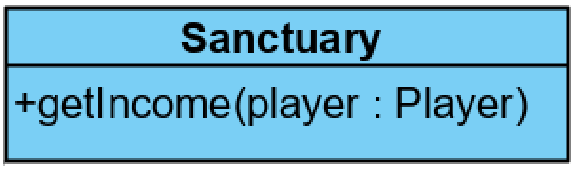


Figure 21: Sanctuary Class Diagram

It is the child class of the structure and it’s one of the structures in the game.

4.2.15.1 Methods of Sanctuary Class

public getIncome(Player player) :It adds income to the players income where every structure have different income number.

4.2.15 Stronghold Class

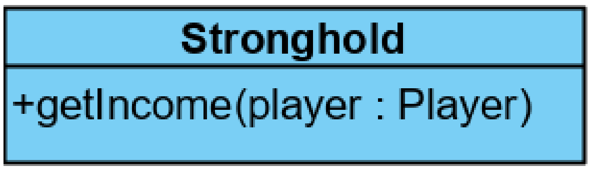


Figure 22: Stronghold Class Diagram

It is the child class of the structure and it’s one of the structures in the game.

4.2.15.1 Methods of Stronghold Class

public getIncome(Player player) :It adds income to the players income where every structure have different income number.

4.2.16 Tile Abstract Class

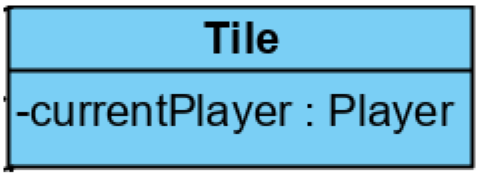


Figure 23: Tile Class Diagram

4.2.16.1. Attributes of Tile Class

private Player currentPlayer: It keeps track of the current player(player whose turn it is) in the game.

4.2.17 ScoringTile Class

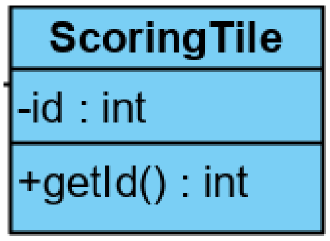


Figure 24: ScoringTile Class Diagram

4.2.17.1 Attributes of ScoringTile Class

private int id : It is used to store the scoring tile's id.

4.2.17.2 Methods of ScoringTile Class

public int getId(): It returns the id number of the scoring tile.

4.2.18 ScoringTileList Class

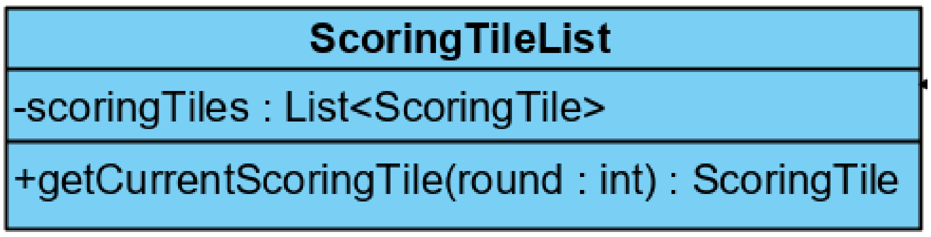

ScoringTileList
-scoringTiles : List<ScoringTile>
+getCurrentScoringTile(round : int) : ScoringTile

Figure 25: ScoringTileList Class Diagram

It is used to get all the scoringTile classes into an ArrayList.

4.2.18.1 Attributes of ScoringTileList Class

private List<ScoringTile> scoringTiles: It is used to store all the scoring tiles to an array list.

4.2.18.2 Methods of ScroingTileList Class

public ScoringTile getCurrentScoringTile(int round): It returns the scoring tile depending on the round number.

4.2.19 FavorTile Class

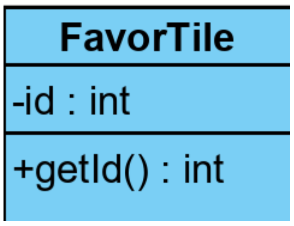


Figure 26: FavorTile Class Diagram

It represents the favor tiles in the game.

4.2.19.1 Attributes of FavorTile Class

private int id: It is used to store the id of favor tile

4.2.19.2 Methods of FavorTile Class

public int getId() : It is used to return the id of the favor tile.

4.2.20 FavorTileList Class

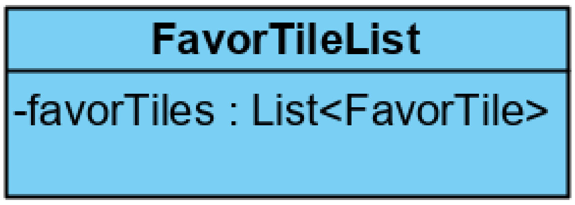


Figure 27: FavorTileList Class Diagram

It is used to store all the favor tiles in the game.

4.2.20.1 Attributes of FavorTileList Class

private List<FavorTile> favorTiles : It is used to store all the favor tiles into an array list.

4.2.21 TownTile Class

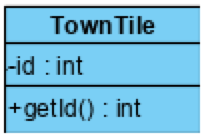


Figure 28: TownTile Class Diagram

4.2.22 TownTileList Class

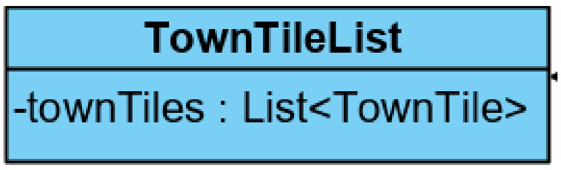


Figure 29: TownTileList Class Diagram

4.2.22.1 Attributes of TownTileList Class

private List<TownTile> townTiles: It is used to store all the town tiles into an array list.

4.2.23 BonusCard Class

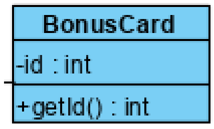


Figure 30: BonusCard Class Diagram

4.2.23.1 Attributes of BonusCard Class

private int id : It stores the id of the bonus card.

4.2.23.2 Methods of BonusCard Class
public int getId(): It returns the id of the bonus card class.

4.2.24 BonusCardList Class

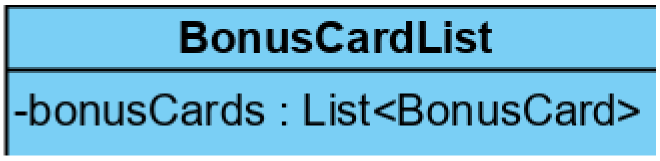
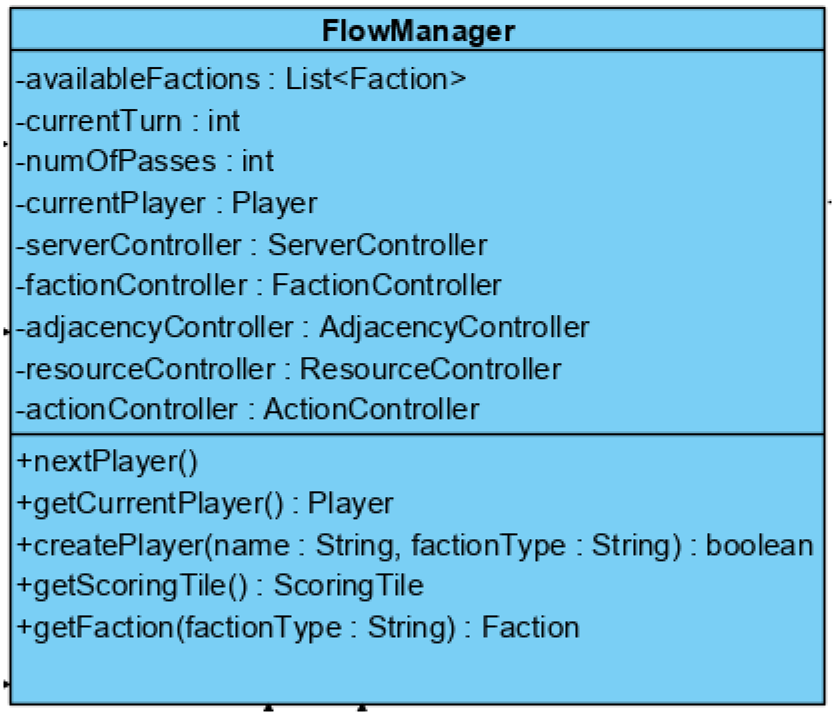


Figure 31: BonusCardList Class Diagram

4.2.24.1 Attributes of BonusCardList Class

private List<BonusCard> bonusCards: It is used to store all the bonus cards into an array list.

4.2.25 FlowManager Class


FlowManager
-availableFactions : List<Faction>
-currentTurn : int
-numOfPasses : int
-currentPlayer : Player
-serverController : ServerController
-factionController : FactionController
-adjacencyController : AdjacencyController
-resourceController : ResourceController
-actionController : ActionController
+nextPlayer()
+getCurrentPlayer() : Player
+createPlayer(name : String, factionType : String) : boolean
+getScoringTile() : ScoringTile
+getFaction(factionType : String) : Faction

4.2.25.1 Attributes of FlowManager Class

private List<Faction> availableFactions : It is used to store all the available factions in the game.

private int currentTurn : It is used to store the current turns number.

private int numOfPasses : It is used to store the number of passes in the round.

private Player currentPlayer : It stores the player object whose round is now.

4.2.25.2 Methods of FlowManager Class

public void nextPlayer() : When the currentPlayer is done with the turn and used actions, the nextPlayer() function is called to get the next player .

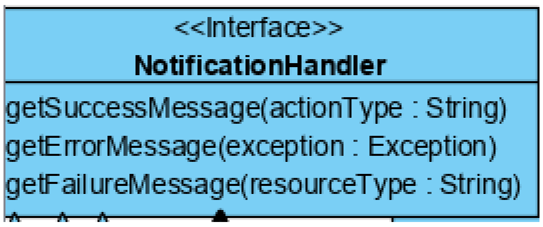
public Player getCurrentPlayer() : It is used to get the currentPlayer object.

public void createPlayer(String name,String factionType): It is called when the game starts. Initially game class calls it to create the desired number of players one by one.

public ScoringTile getScoringTile() : It gets the current scoring tile of the round.

public Faction getFaction(String factionType): It returns the Faction instance for the given faction type name.

4.2.26 NotificationHandler Interface


<<Interface>>
NotificationHandler
getSuccessMessage(actionType : String)
getErrorMessage(exception : Exception)
getFailureMessage(resourceType : String)

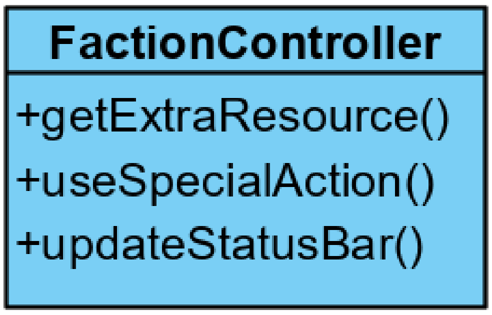
4.2.27.1 Methods of NotificationHandler Interface

public void getSuccessMessage(String actionType) : Returns a success message for notification after a successful action.

public void getErrorMessage(Exception exception) : Returns an error message for notification after an exception.

public void getFailureMessage(String resourceType) : Returns a failure message for notification after a failing action because of inadequate resources.

4.2.27 FactionController Class



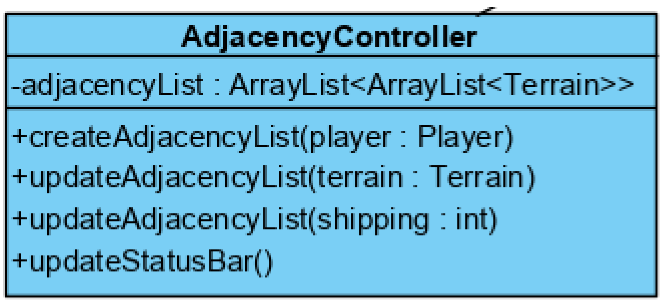
4.2.27.1 Methods of FactionController Class

public void getExtraResource(): Obtains extra resources which are specific for each faction.

public void useSpecialAction(): Uses special action type which is specific for each faction.

public void updateStatusBar() : Updates the status bar to give notification about the current status of a faction.

4.2.28 AdjacencyController Class


AdjacencyController
-adjacencyList : ArrayList<ArrayList<Terrain>>
+createAdjacencyList(player : Player)
+updateAdjacencyList(terrain : Terrain)
+updateAdjacencyList(shipping : int)
+updateStatusBar()

4.2.28.1 Attributes of AdjacencyController Class

private ArrayList<ArrayList<Terrain>> adjacencyList : It is used to store the adjacency list.

4.2.28.2 Methods Of AdjacencyController Class

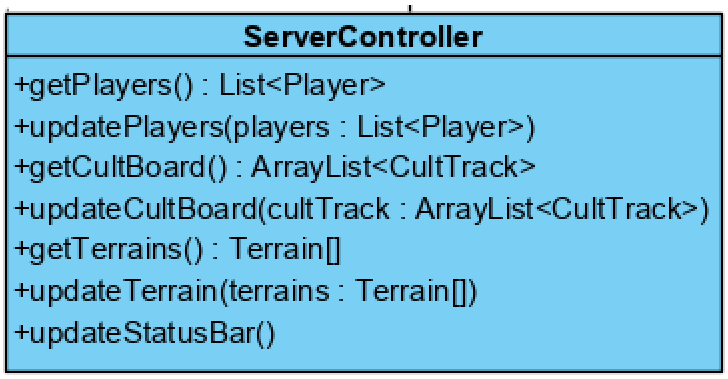
public void createAdjacencyList(Player player) : It creates the adjacency list for the input player object.

public void updateAdjacencyList(Terrain terrain): It updates the adjacency list by looking the terrains.

public void updateAdjacencyList(int shipping): It updates the adjacency list by looking the shipping level to adapt adjacency to water tiles.

public void updateStatusBar() : Updates the status bar to give notification about the current status of actions in the class.

4.2.29 ServerController Class


ServerController
+getPlayers() : List<Player>
+updatePlayers(players : List<Player>)
+getCultBoard() : ArrayList<CultTrack>
+updateCultBoard(cultTrack : ArrayList<CultTrack>)
+getTerrains() : Terrain[]
+updateTerrain(terrains : Terrain[])
+updateStatusBar()

4.2.29.2 Methods of ServerController Class

public List<Player> getPlayers(): It returns the players list which are in the server

public void updatePlayers(List<Player> players):

public ArrayList<CultTrack> getCultBoard(): It returns the current cult board.

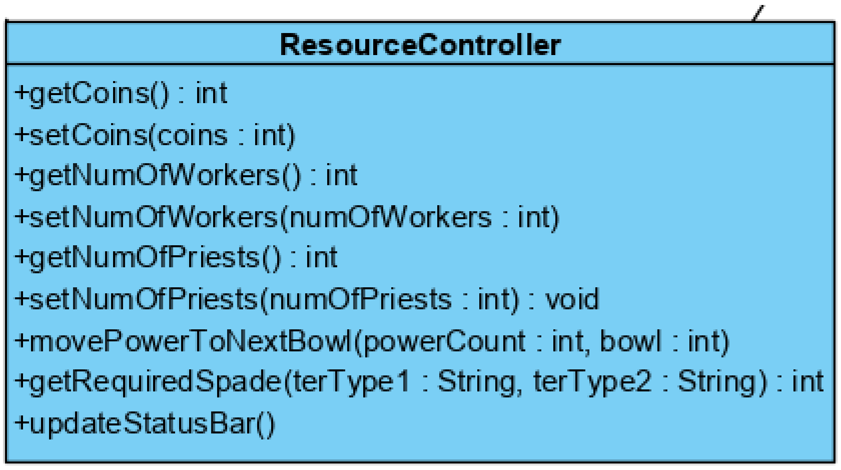
public updateCultBoard(ArrayList<CultTrack> cultTrack): This method replaces the new cult track with the old one.

public Terrain[] getTerrains(): It returns the terrain array.

public void updateTerrain(Terrain[] terrain): It updates the terrain array with the new terrain array.

public void updateStatusBar(): Updates the status bar to give notification about the current status of the server.

4.2.30 ResourceController Class


ResourceController
+getCoins() : int
+setCoins(coins : int)
+getNumOfWorkers() : int
+setNumOfWorkers(numOfWorkers : int)
+getNumOfPriests() : int
+setNumOfPriests(numOfPriests : int) : void
+movePowerToNextBowl(powerCount : int, bowl : int)
+getRequiredSpade(terType1 : String, terType2 : String) : int
+updateStatusBar()

4.2.30.1 Methods of ResourceController Class

public int getCoins(): Gets the total number of coins.

public void setCoins(int coins): Sets the number of coins of a specific player.

public int getNumOfWorkers(): Gets the total number of workers.

public void setNumOfWorkers(int numOfWorkers): Sets the total number of workers.

public int getNumOfPriests(): Gets the total number of priests.

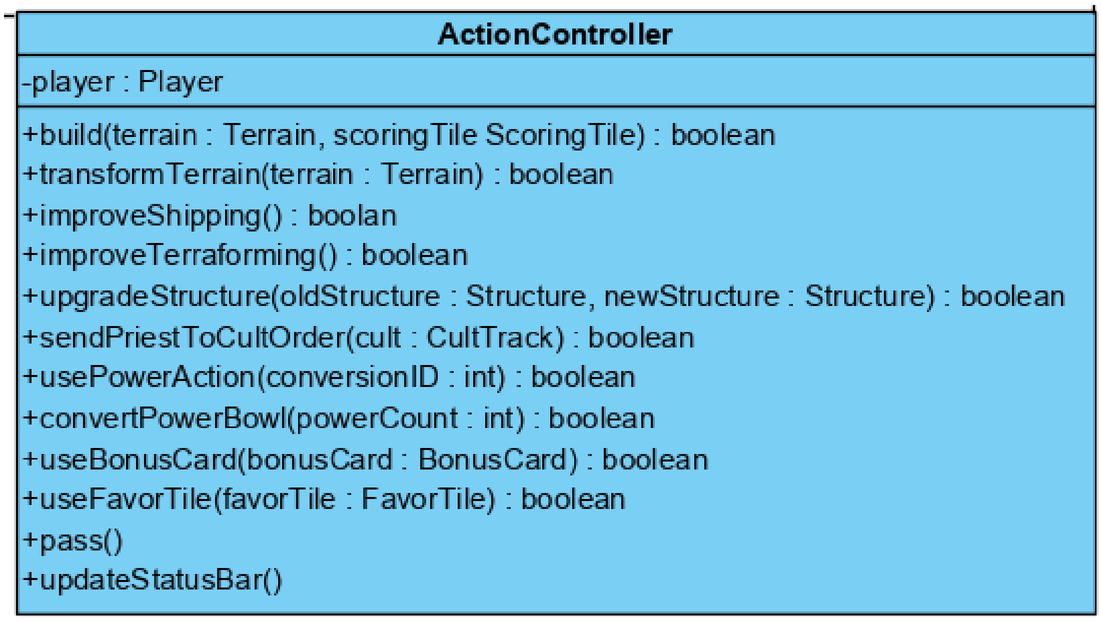
public void setNumOfPriests(int numOfPriests): Sets the total number of priests.

public void movePowerToNextBowl(int powerCount, int bowl): It moves given number of power tokens from indicated bowl to next bowl.

public static int getRequiredSpade(String terType1,String terType2) : Returns required number of spades for transformation from one terrain type to another one.

public void updateStatusBar() : Updates the status bar to give notification about the current status of actions.

4.2.31 ActionController Class



```
ActionController
-player : Player
+build(terrain : Terrain, scoringTile ScoringTile) : boolean
+transformTerrain(terrain : Terrain) : boolean
+improveShipping() : boolan
+improveTerraforming() : boolean
+upgradeStructure(oldStructure : Structure, newStructure : Structure) : boolean
+sendPriestToCultOrder(cult : CultTrack) : boolean
+usePowerAction(conversionID : int) : boolean
+convertPowerBowl(powerCount : int) : boolean
+useBonusCard(bonusCard : BonusCard) : boolean
+useFavorTile(favorTile : FavorTile) : boolean
+pass()
+updateStatusBar()
```

4.2.31.1 Methods of ActionController Class

public boolean build(Terrain terrain, ScoringTile scoringTile): If the terrain is empty and the player has enough resources, it builds dwelling on the given terrain. Then, gets the income both from the structure and the scoring tile if it is possible.

public boolean transformTerrain(Terrain terrain): Transforms the terrain if it is empty and the player has enough resources.

public boolean improveShipping(): Improves shipping skill by using required resources.

public boolean improveTerraforming(): Improves terraforming skill by using required resources.

public boolean upgradeStructure(Structure oldStructure, String newStructure): Upgrades old structure to the new structure by using required resources. Then, it gets the income from the new structure.

public boolean sendPriestToCult(int cultID): Sends priest to one of the cults' order in order to advance on that cult track.

public boolean usePowerAction(int powerActionID): Uses power tokens in bowl 3 and moves those tokens to bowl 1 in order to have coins, workers, priests, spades or bridges.

public boolean convertPowerBowl(int powerCount): Moves power tokens from bowl 2 to bowl 3 by sacrificing power tokens.

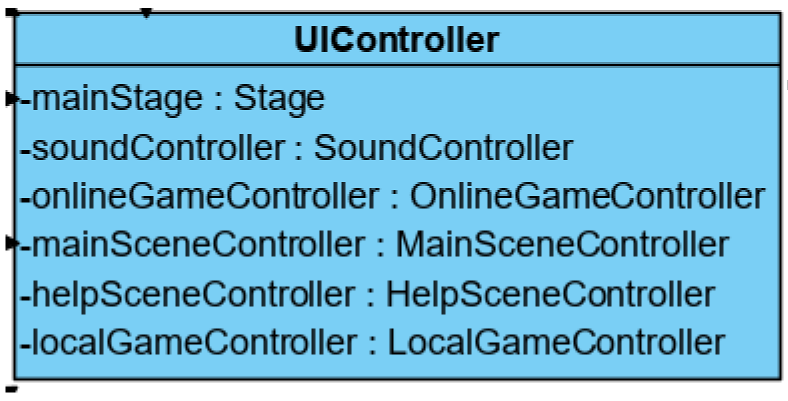
public boolean useBonusCard(BonusCard bonusCard): Uses given bonus card accordingly.

public boolean useFavorTile(FavorTile favorTile): Uses given favor tile accordingly.

public void pass(): Passes and waits for the other players to pass.

public void updateStatusBar() : Updates the status bar to give notification about the current status of actions.

4.2.32 UIController Class



UIController
-mainStage : Stage
-soundController : SoundController
-onlineGameController : OnlineGameController
-mainSceneController : MainSceneController
-helpSceneController : HelpSceneController
-localGameController : LocalGameController

private Stage mainStage : It is used to store the Stage variable.

private SoundController soundController: It is used to store

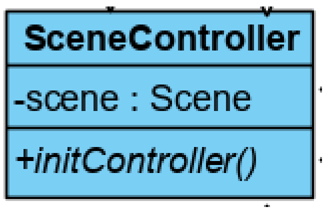
private OnlineGameController onlineGameController:

private MainSceneController mainSceneController:

private HelpSceneController helpSceneController:

private LocalGameController localGameController:

4.2.33 SceneController Class



4.2.33.1 Attribute of SceneController Class

private Scene scene: It is used to store the Scene.

4.2.33.2 Methods of SceneController Class

public void initController(): Initializes the scene controller.

4.2.34 SoundController Class

SoundController

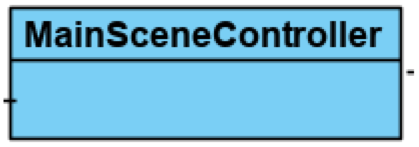
This class is created to control the sounds in the UI class.

4.2.35 OnlineGameController Class

OnlineGameController

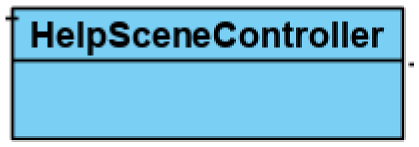
Controller class for the online game mode.

4.2.36 MainSceneController Class



Controller class for the Main Scene of the game.

4.2.37 HelpSceneController Class



Controller class for the Help Scene of the game.

4.2.38 LocalGameController Class

Controller class for the Local Play mode of the game.

4.3. Packages

We named our packages according to our Subsystem Diagram Package structure. Thus, packages have the same naming conventions with the subsystems. Consequently, packages allow us to conduct tests on entire subsystems using frameworks like JUnit [3]. Moreover, packages provide us with access control between subsystems.

4.4. Class Interfaces

We have NotificationHandler interface being implemented by all the control classes. The interface is used to notify the player about the result of the control that has been made. Therefore, the player will be notified whether the action they took is successful or not. Additionally, unsuccessful actions will be presented to the player with the reason of failure.

4.5. Design Patterns

4.5.1. Facade Design Pattern

We have FlowManager and Game classes as facade classes. All controller classes except those ones for the UI, communicate with entity objects via FlowManager class. Similarly, All changes happening in the entities are transmitted to FlowManager class via Game class. By doing so, we can separate subsystems into two layers as controller and model.

Likely, all changes happening in the entities are transmitted to GameEngine class via Game class.

4.5.2. Singleton Design Pattern

We have controller classes and Game class as singletons. To clarify, all entities reside in one instance of the Game class and all controller classes control all their subjects from the beginning to the end of the game. By having so, we have greater access to our controllers and model as their instances can globally be accessible and we can be sure that any update will be visible throughout the system.

5. Improvement Summary

1. Subsystem decomposition is improved.
    - Names are more coherent.
    - New packages are introduced.
    - The system is partitioned more deeply.
2. Object Design trade-offs are updated.
    - Concerns regarding complexity and our standing are clarified and explained in more detail.
3. Design patterns are introduced.
4. Object Design is updated to represent the design patterns.
5. Control classes are divided in a way allowing delegation.

6. References

1. javadoc
2. Testwell CMTJava
3. JUnit 5