BILKENT UNIVERSITY

Department Of Computer Science

CS319 - Object-Oriented Software Engineering

Design Report

Game: Terra Mystica

Group: 1E

Berdan Akyürek 21600904

Ege Hakan Karaağaç 21702767

Ömer Olkun 21100999

Aziz Ozan Azizoğlu 21401701

Fırat Yönak 21601931

1. Introduction

1.1 Purpose of the system

The purpose of Terra Mystica game is to provide users an easy and fun space where they can play their favorite board game much quicker and easier than the real one. To provide this functionalities its design is implemented in a way that can make Terra Mystica portable, user-friendly and fast.

1.2 Design goals

Design goals are mostly based on the non-functional requirements that were already discussed in the analysis report. Non-functional requirements are going to be discussed in more detail with the following design goals that are described below.

1.2.1 Criteria

Performance: Performance is a very important goal for our game. For players, performance is the most significant feature; that's why, our game will provide pleasant time for players while they are playing Terra Mystica. To increase the performance of our game,the game will be implemented with using Javafx. After implementation process is done, we will review each code parts to make more efficient. If any bug or mistake is found, it will be fixed. Our aim is to provide more satisfactory time for players.

Modifiability: Terra Mystica is based on object oriented programming language that enables designers to change its structure after the end of the implementation. Implementation will be done to make no strong connection between classes. With that new features can be adapted to game without any huge alterations on other classes; that's why, Terra will be easy modifiable game at the end of implementation.

Portability: Portability is one of the most important feature of a game because number of total players is the most significant thing for developers. To reach more players, a game should run on different machines and platforms in order to provide this feature, Java

will be used. With the usage of Java, the game will be accessible from all machines which have JVM.

User Experience: Terra Mystica is a fast paced game that requires the users to keep track of various elements and factors. In order for user will have good experiences while they are playing game, we will take understandability and simplicity into consideration. Additionally, to make things easier for players, how to play part will be available for players. When they want to see rules of the game, they can reach to the page both from menu and in game.

Extendibility: In the implementation process, the possibility of changes and improvements which can be made on code will be considered. First releases of almost each game or program needs to be improved because of its bugs, lacks; that's why, we take this into consideration and try to have a design that is open to be extended. We try to separate details and keep them in different parts. In that manner we use an object-oriented design pattern. Using inheritance we try to keep the abstract components and customised components in different levels so that when an improvement or fix is necessary, we will know where the problem is and how to do it. It also provides us to think just on specified parts.

Reusability: Some of the classes of Terra Mystica can be used in other projects. All of the games are designed for usage of players; that's why, Player class may be a starting point for future games. Many of games user interface has same bases; that's why, the Menu, How To Play and Credits class can be beneficial for new games.

2.High-level Software Architecture

2.1 Subsystem decomposition

Terra Mystica is divided into some subsystems. It is done by putting related classes and components into same subsystem. Decomposition is done in order to reduce complexity and to gain low coupling and high coherence. We used the MVC (Model-View-Controller) architecture in our game; that's why, the components of Terra Mystica are Model, View and Controller.

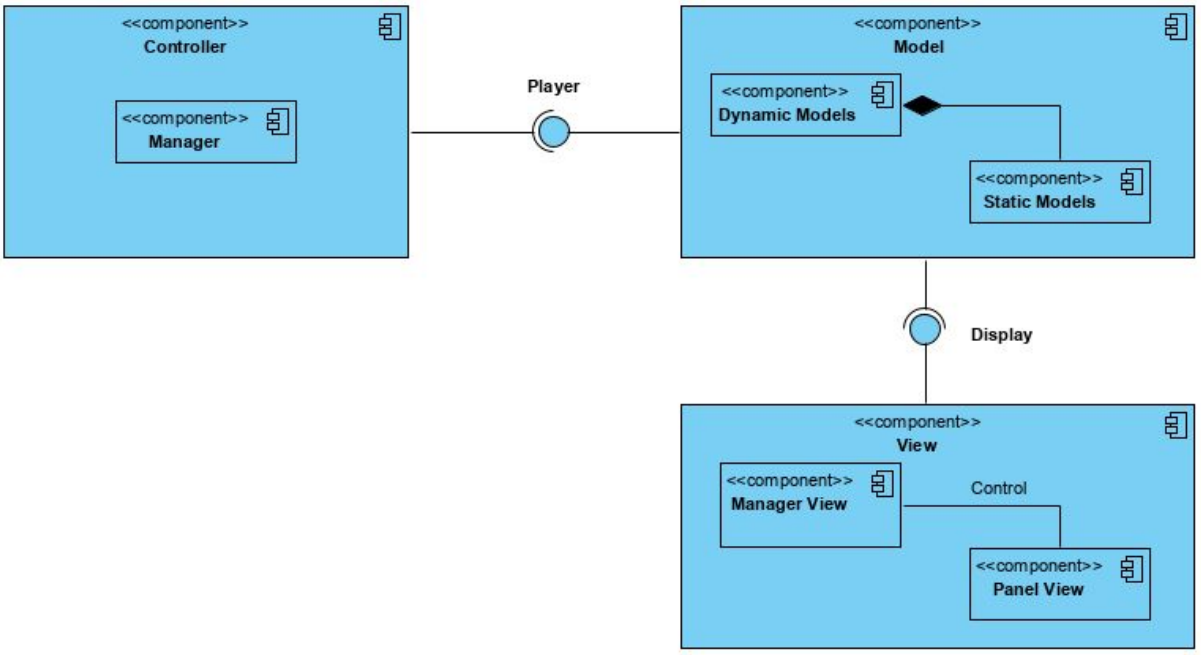

Figure 1: Subsystem decomposition diagram of the game

2.2 Hardware/Software Mapping

Terra Mystica game will be implemented in Java programming language. Through the implementation process, we will benefit from Javafx libraries; that's why, we need a Java Runtime Environment to both work on and execute and a Java Development Kit which includes both Java and Javafx libraries. As hardware requirement, Terra Mystica I/O devices like a mouse and a display

2.3 Persistent Data Management

In Terra Mystica there is not a feature to save the game and continue it from saving point. Also, it will not be a web based game. Because of we do not need to use complicated database or cloud data management. In order to store data from current game, system will use its own model classes; that's mean our data will be stored locally.

2.4 Access Control and Security

Terra Mystica will not use any network, internet or database connection. Players current data will be stored locally and it will not be long term,. When they close the game, the data of last game will be deleted. Because of that there is no need to access control or security check.

2.5 Boundary conditions

- **Installation:** To execute Terra Mystica, a Java Archive (jar) file will be used; that's why, installation will not be a need to play Terra Mystica. In order to execute jar files to play game java environment is the only need.
- **Initialization:** For initialization, Terra Mystica does not require anything since the game will not use any database or file to record something. If there is an opened game, player cannot open a new one.
- **Termination:** Terra Mystica can be terminated by clicking the "Exit" button from at the right top of title bar. There will be also a button on main menu to exit from the game. When the game is in fullscreen, the title bar will be hidden. Firstly, players need to click "Esc" to see title bar at the top of screen. After that they can easily click the "Exit" button in order to terminate gamme.
- **Failure:** When the game fails, their will be some functionality issues. If the game collapses due to a failure, no data will be saved from last game and players needs to start from beginning. At the time game fails, terminate the game and restart it can solve the functionality problems.

3. Subsystem services

3.1 Model

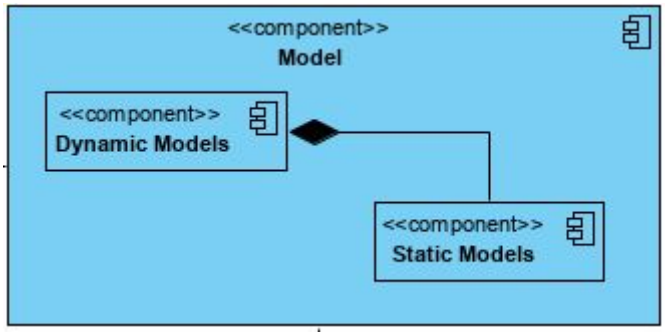

Figure 2: The diagram of model subsystem of Terra Mystica

This subsystem contains the game objects. It make its action by the messages of Controller. When Model receives a directive from Controller, it makes alterations and records them. If any updates is necessary on View, Model will detect it and delivers a message to View about the alterations which it needs to be done. Model subsystem is composed of two components: Dynamic and Static models.

- **Dynamic Models:** In dynamic models component, there are dynamic objects. Dynamic objects are objects which change states during the game. Such as Player.
- **Static Models:** In static models component, there are static objects These objects are the ones which remain in the same state during the game. Examples of these are Faction, BonusCards.

3.2 View

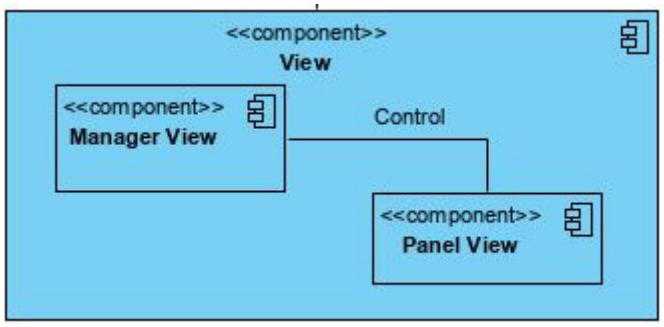

Figure 3: The diagram of view subsystem of Terra Mystica

This subsystem displays the user interface of whole system depending the messages of Model. It is the component of Terra Mystica which is responsible to interact users and gets inputs from them. If any directives comes from Model, it will be updated. View is composed of Manager View and Panel View.

- **Manager View:** It is the component of View which includes class to update panels view classes.
- **Panel View:** This component includes all classes which are responsible from all screens like main menu, how to play and game screen.

3.3 Controller

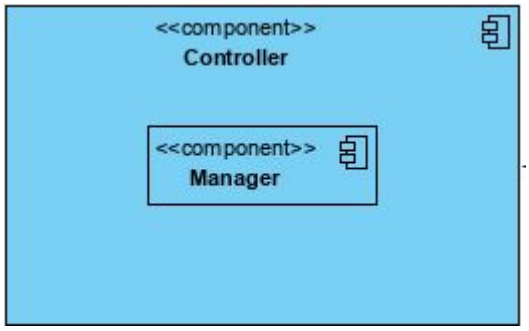

Figure 4: The diagram of controller subsystem of Terra Mystica

This subsystem's responsibility is to control the game and deal with user inputs. It is the component which receives inputs from users. When it gets messages from users, it will send messages to Model component by the contents of receiving inputs. It has only one component.

- **Manager:** This component responsible to initialize the game and handling the game sequences which are not initiated by user inputs. Also, it sends messages to model subsystem according to what should happen in the game.

4. Low-level design

4.1 Object Design Trade-offs

Memory versus Performance: In our game, we have created many classes and used their objects in many places. Therefore we were able to have a more object-oriented design. Our main goal is to get more performance from the game since players expects to take all joy from the game, they can; that's why, we gave more weight on performance rather than memory. Our purpose is not to increase memory with unnecessary codes;however, each feature needs more coding and each new coding results with more memory. As a result of that our priority is performance.

Simplicity versus Features: When a new game is released, each player's first wonder mostly is the features of a game since every new feature of game meaning is a new thing to do in the game; however, every feature adds more complexity to game; that's why, it is a challenging problem to decide. We prefer to add more features to game rather than making Terra Mystica simpler. Because of these reasons, our priority is features instead of complexity.

4.2 Final Object Design

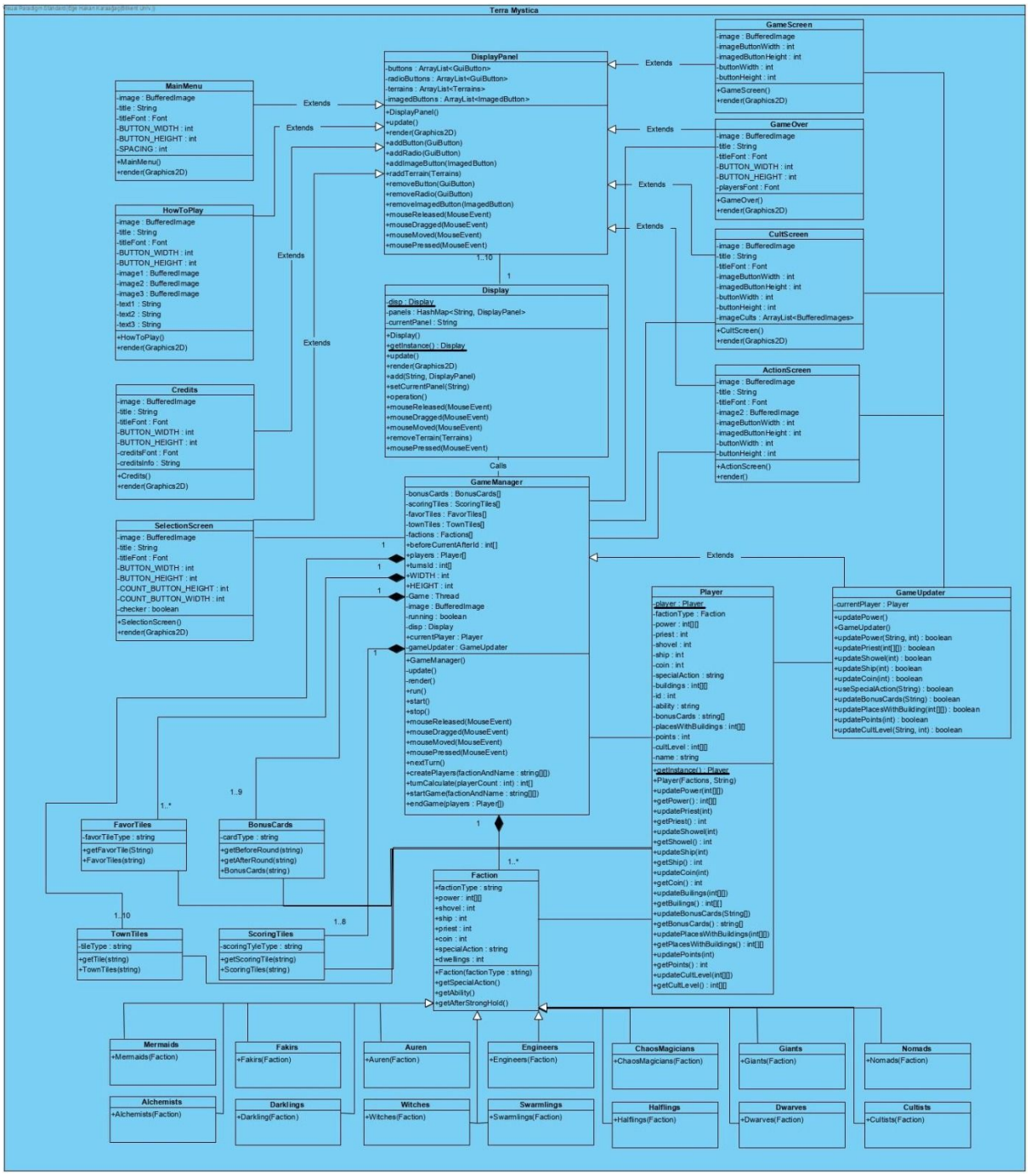

Figure 5: The class diagram of Terra Mystica

4.2.1 Immutable Design Pattern

In the low level design, there will be some instances which will not be changed during the game. To recognize instances, they need to be specified differently from others. In order

to do that for all of the implementation process, Immutable Design Pattern will be used in our classes. This pattern is used in both view and controller component.

4.2.2 Facade Design Pattern

In the low level design, the Facade Design Pattern will be used. With this design pattern, we will have abstract access from a layer to another one. This pattern prevent direct access between the layers. Player and Display are facade classes of Terra Mystica.

4.3 Packages

4.3.1. Internal Library Packages

Model Package

This package includes classes which are responsible to manage internal game objects and their storage.

View Package

This package includes the classes which are responsible to manage user interface.

Controller Package

This package includes the classes which are responsible to manage the system.

4.3.2. External Library Packages

- **javafx.scene.layout**

This package provides User Interface layouts to fit GUI objects. We used different classes of this package to manage UI components.

- **javafx.event**

Provides basic framework for FX events, their delivery of inputs between different hardware components and handling.

- **javafx.scene.input**

Provides the set of classes for mouse and keyboard input event handling. We used this package to handle MouseEvents.

- **javafx.scene.image**

Provides the set of classes for loading and displaying images. We use images to display cards, icons of items and wonder boards.

4.4 Class Interfaces

4.4.1 Controller Class Interfaces

Our controller interface consists of two classes which are called GameManager and GameUpdater. This class is responsible of every mouse action, updating players attributes, starting the game, ending the game, changing rounds, update user interface when any action occurred.

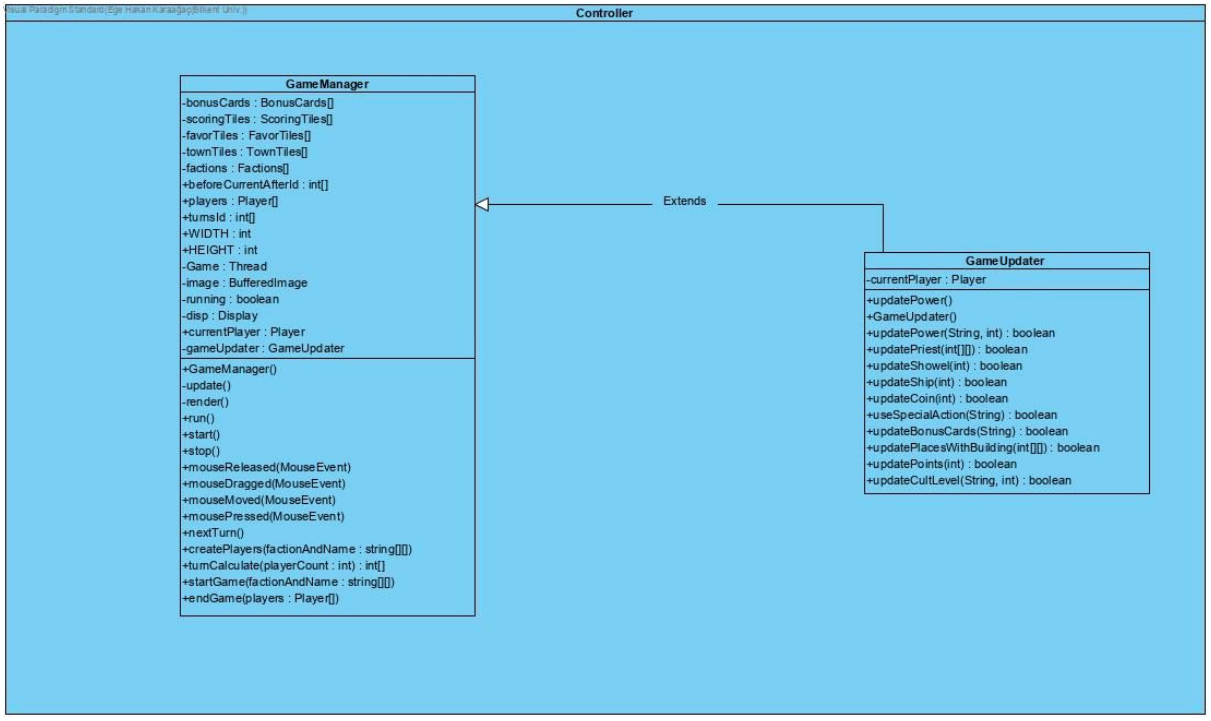

Figure 6: The diagram of Controller Subsytem

GameManager Class:

Figure 7: GameManager Class

This class is responsible to manage the game.

Attributes:

- **private final BonusCards[] bonusCards:** This attribute provides an array of bonus card objects.
- **private final ScoringTiles[] scoringTiles:** This attribute provides an array of scoring tile objects.
- **private final Favortiles[] favorTiles:** This attribute provides an array of favor tile objects.
- **private final TownTiles[] townTiles:** This attribute provides an array of town tile objects.

- **private final Factions[] factions:** This attribute provides an array of factions of characters in the game.
- **private int[] beforeCurrentAfterId:** This attribute holds the ids of players in the previous turn, the current turn and the next turn.
- **private Player[] players:** This attribute holds an array of player objects in a game session.
- **private int[] turnsId:** This attribute holds players' ids aligned according to order of turns.
- **final int WIDTH:** This attribute is the width of the screen.
- **final int HEIGTH:** This attribute is the height of the screen.
- **Thread Game:** It is the thread of the game.
- BufferedImage image: This attribute holds the background image of game
- **private boolean running:** This attribute holds information about whether game is running or not.
- **private Display disp:** This attribute enables GameManager to reach attributes and functions of Display and communicate with the class.
- **private Player currentPlayer:** This attribute is to communicate with player and reach its methods and attributes.
- **private GameUpdater gameUpdater:** This attribute enables GameManager to reach attributes and functions of GameUpdater and communicate with the class.

Constructor:

- **GameManager():** Constructor for the GameManager.

Methods:

- **public void update():** This method will be used to update the game by actions of players.
- **public void render():** It is the method which is responsible to create user interface and renew current ones.
- **public void start():** This method will be used to start the game and necessary objects.
- **public void stop():** It is the method which is used to create user interface of the game.
- **public void mouseReleased(MouseEvent):** This method will be invoked whenever the mouse is released.

- **public void mouseDragged(MouseEvent):** This method will be invoked whenever the mouse is dragged.
- **public void mouseMoved(MouseEvent):** This method will be invoked whenever the mouse is moved.
- **public void mousePressed(MouseEvent):** This method will be invoked whenever the mouse is pressed.
- **public void nextTurn():** This method passes the move turn to next player.
- **public void createPlayers(String[][] factionAndName):** This method will be invoked in the start of the game to create players with their factions and names.
- **public int[] turnCalculate(int playerCount):** This method determines the order of turns according to number of players.
- **public void startGame(String[][] factionAndName):** This method will be invoked to start the game session.
- **public void endGame(Player[] players):** This method will be invoked to end the game session.

GameUpdater Class:

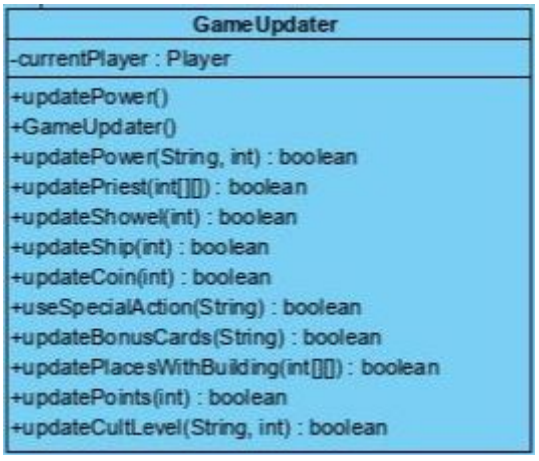

Figure 8: GameUpdater Class

This class is responsible to update the current player's by its actions.

Attributes:

- **private Player currentPlayer:** It is the attribute which holds the information of current player.

Constructor:

- **GameUpdater():** Constructor for GameUpdater class.

Methods:

- **public boolean updatePower(String,int):** This method is used to update player's power.
- **public boolean updatePriest(int[][]):** This method is used to update current player priest according to his or her actions.
- **public boolean updateShowel(int):** This method is used to update current player showel according to his or her actions.
- **public boolean updateShip(int):** When current player makes an action which affects his or her ship, this method updates the player's ships.
- **public boolean updateCoin(int):** This method is used to update current player coins according to player's actions.
- **public useSpecialAction(String):** When current player use special action this method updates the player's special actions
- **public boolean updateBonusCards(String):** When current player play a bonus cards, this methods updates player's bonus cards.
- **public boolean updatePlacesWithBuilding():** When current player makes an action on buildings or construct a new one, this method updates player's buildings.
- **public boolean updatePoints(int):** At the end of each round player's points are updated according to their advancements and coins by this method.
- **public boolean updateCultLevel(String, int):** This method is responsible to update current player's cult according to his or her actions.

4.4.2 View Class Interfaces

This section of the games class diagram is responsible of each and every gui in the game. It has the mainmenu, how to play, credits, selection, game, gameover, cult, action screens. With the commands of game manager class it updates the screen. According to players features and turns the game screen is updated with every move.

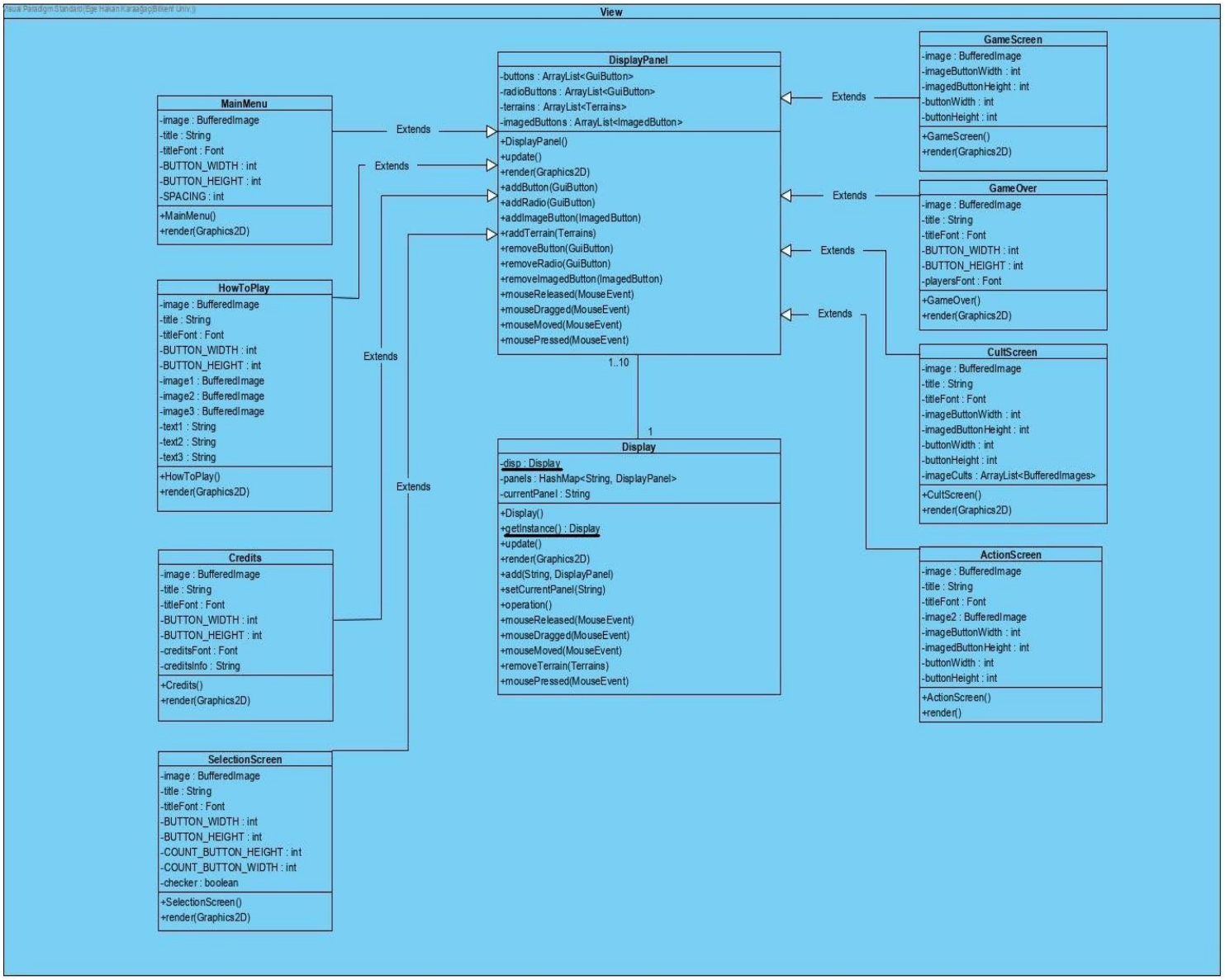

Figure 9: View Diagram

Display Class:

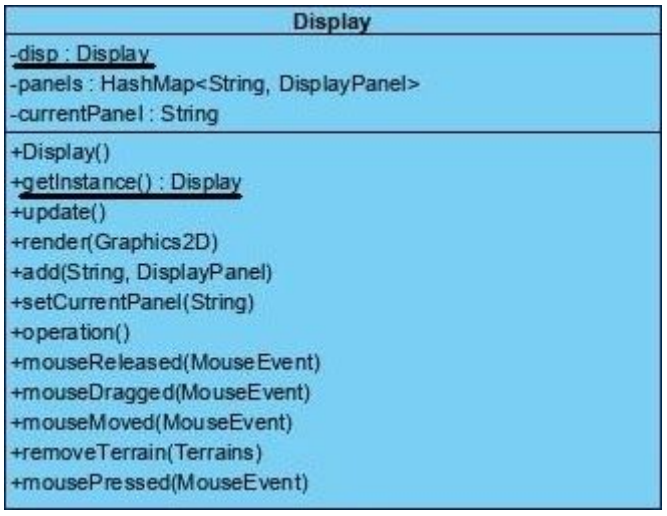

Figure 10: Display Class

This class calls other display classes

Attributes:

- **private Display display :** this attribute controls all the screens.
- **private HashMap<String, DisplayPanel> :** List of all screens.
- **private String currentPanel :** the current panel in strings.

Methods:

- **public Display() :** constructor of display class.
- **public void getInstance() :** return the display.
- **public void update() :** updates the current screen.
- **public void render(Graphics2D) :** renders the current screen.
- **public void add(String, DisplayPanel) :** adds a screen to array.
- **public void setCurrentPanel(String) :** sets the current panel.
- **public void mouseReleased(MouseEvent) :** mouse event shows that mouse is released. Calls the same method of currentScreens.
- **public void mousePressed(MouseEvent) :** mouse event shows that mouse is pressed. Calls the same method of currentScreens.

- **public void mouseDragged(MouseEvent) :** mouse event shows that mouse is dragged. Calls the same method of currentScreens.
- **public void mouseMoved(MouseEvent) :** mouse event shows that mouse is moved. Calls the same method of currentScreens.

DisplayPanel Class:

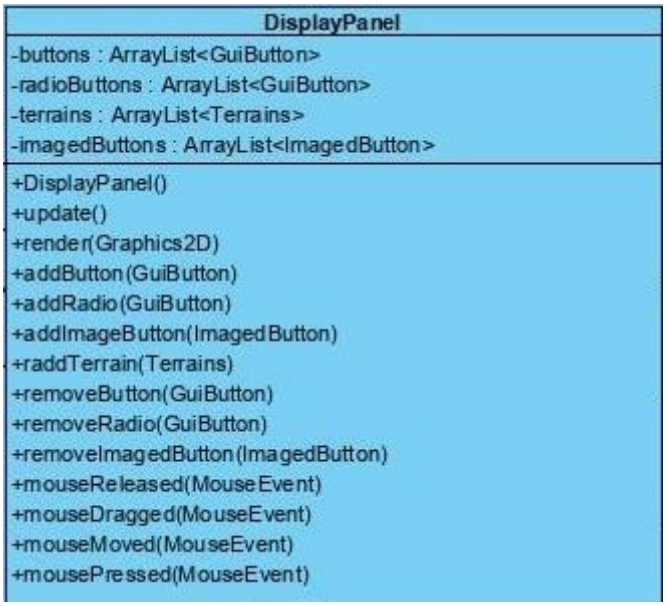

Figure 11: DisplayPanel Class

This class is extended by all the screens.

Attributes:

- **private ArrayList<GuiButton> buttons :** array of buttons.
- **private ArrayList<GuiButton> radioButtons :** array of radio buttons.
- **private ArrayList<Terrains> buttons :** array of terrains.
- **private ArrayList<ImagedButton> buttons :** array of imaged buttons.

Methods:

- **public DisplayPanel() :** constructor of display panel class.
- **public void update() :** updates the current screen.

- **public void render(Graphics2D) :** renders the current screen.
- **public void addButton(GuiButton) :** adds button to array.
- **public void addRadio(GuiButton) :** adds radio button to array.
- **public void addTerrain(Terrains) :** adds terrain to array.
- **public void addImagedButton(ImagedButton) :** adds imaged button to array.
- **public void removeButton(GuiButton) :** removes button from array.
- **public void removeRadio(GuiButton) :** removes radio button from array.
- **public void removeTerrain(Terrains) :** removes terrain from array.
- **public void removeImagedButton(ImagedButton) :** removes imaged button from array.
- **public void mouseReleased(MouseEvent) :** mouse event shows that mouse is released.
- **public void mousePressed(MouseEvent) :** mouse event shows that mouse is pressed.
- **public void mouseDragged(MouseEvent) :** mouse event shows that mouse is dragged.
- **public void mouseMoved(MouseEvent) :** mouse event shows that mouse is moved.

Main Menu Class:

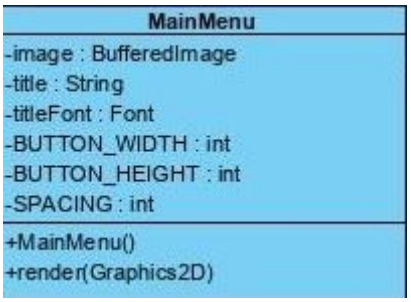

Figure 12: Main Menu Class

This class shows main menu with buttons. User can quit, start a game, learn how to play and see the credits.

Attributes:

- **private BufferedImage image :** Image of background photo.
- **private String title :** title.
- **private Font titleFont :** titles's font.
- **private final int BUTTON_WIDTH :** buttons width.
- **private final int BUTTON_HEIGHT :** buttons height.
- **private final int SPACING :** spacing count.

Methods:

- **MainMenu() :** constructor of main menu.
- **render(Graphics2D) :** renders all the graphics.

HowToPlay Class:

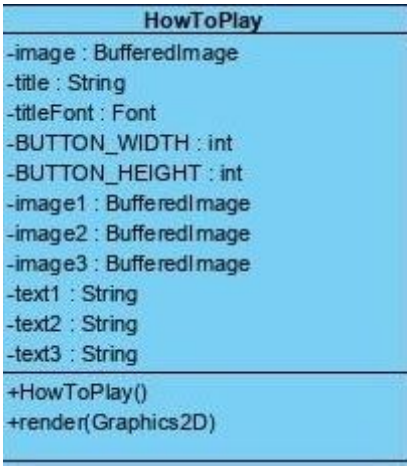

Figure 13:. HowToPlay Class

This class shows how to play with buttons. User can go back and see how to play.

Attributes:

- **BufferedImage image :** Image of background photo.
- **String title :** title.
- **Font titleFont :** titles's font.
- **private final int BUTTON_WIDTH :** buttons width.

- **private final int BUTTON_HEIGHT :** buttons height.
- **private BufferedImage image1 :** how to play description image1.
- **private BufferedImage image2 :** how to play description image2.
- **private BufferedImage image2 :** how to play description image3.
- **private String tex1 :** description of how to play.
- **private String tex2 :** description of how to play.
- **private String tex3 :** description of how to play.

Methods:

- **HowToPlay() :** constructor of how to play.
- **render(Graphics2D) :** renders all the graphics.

Credits Class:

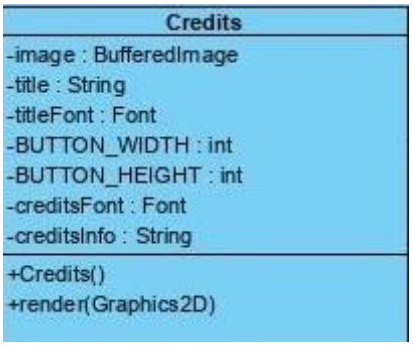

Figure 14: Credits Class

This class shows information about creators of the game.

Attributes:

- **private BufferedImage image :** Image of background photo.
- **private String title :** title.
- **private Font titleFont :** titles's font.
- **private final int BUTTON_WIDTH :** buttons width.
- **private final int BUTTON_HEIGHT :** buttons height.
- **private String creditsInfo :** credits.

- **private Font creditsFont :** credit's font.

Methods:

- **Credits() :** constructor of credits.
- **render(Graphics2D) :** renders all the graphics.

SelectionScreen Class:

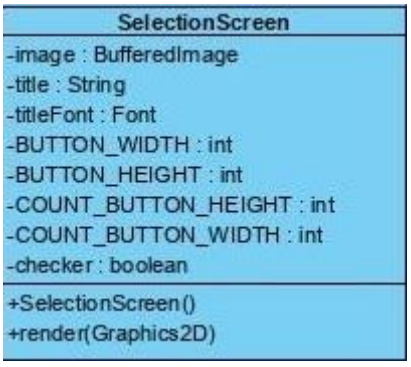

Figure 15: SelectionScreen Class

This class lets players set the number of players, set their names and choose their factions.

Attributes:

- **private BufferedImage image :** Image of background photo.
- **private String title :** title.
- **private Font titleFont :** titles's font.
- **private final int BUTTON_WIDTH :** buttons width.
- **private final int BUTTON_HEIGHT :** buttons height.
- **private final int COUNT_BUTTON_WIDTH :** count buttons width.
- **private final int COUNT_BUTTON_HEIGHT : count** buttons height.
- **private boolean checker :** checks if it is rendered.

Methods:

- **SelectionScreen() :** constructor of selection screen.

- **render(Graphics2D) :** renders all the graphics.

GameOver Class:

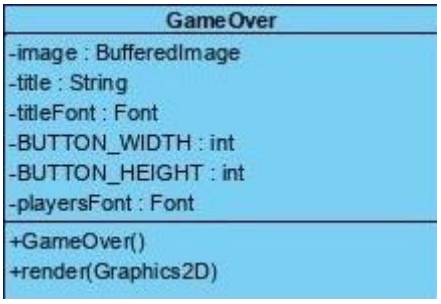

Figure 16: GameOver Class

This class shows information like who is the winner and who has how many points at the end of the game

Attributes:

- **private BufferedImage image :** Image of background photo.
- **private String title :** title.
- **private Font titleFont :** titles's font.
- **private final int BUTTON_WIDTH :** buttons width.
- **private final int BUTTON_HEIGHT :** buttons height.
- **private Font playersFont :** font of players.

Methods:

- **GameOver() :** constructor of game over.
- **render(Graphics2D) :** renders all the graphics.

GameScreen Class:

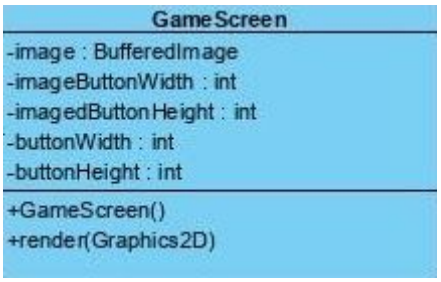

Figure 17: GameScreen Class

This class shows the map and player tabs and buttons to use actions and change to cult screen.

Attributes:

- **private BufferedImage image :** Image of background photo.
- **private int imagedButtonWidth :** imaged buttons width.
- **private int imagedButtonHeight :** imaged buttons height.
- **private int buttonWidth :** buttons width.
- **private int buttonHeight :** buttons height.

Methods:

- **GameScreen() :** constructor of gameScreen.
- **render(Graphics2D) :** renders all the graphics.

CultScreen Class:

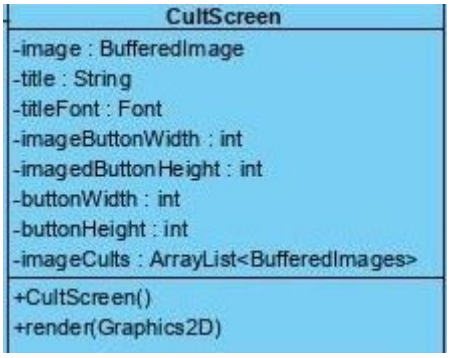

Figure 18: CultScreen Class

This class displays the cults on game screen.

Attributes:

- **private BufferedImage image :** Image of background photo.
- **private String title :** title.
- **private Font titleFont :** titles's font.
- **private int imagedButtonWidth :** imaged buttons width.
- **private int imagedButtonHeight :** imaged buttons height.
- **private int buttonWidth :** buttons width.
- **private int buttonHeight :** buttons height.
- **private ArrayList<BufferedImages> imageCults :** array of cult images.

Methods:

- **CultScreen() :** constructor of cultScreen.
- **render(Graphics2D) :** renders all the graphics.

ActionScreen Class:

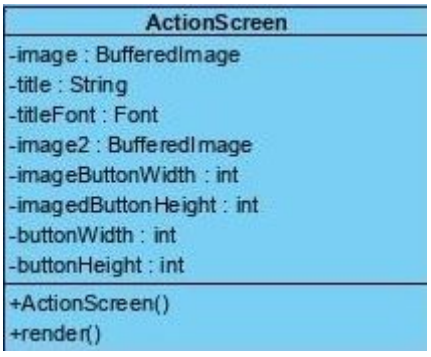

Figure 19: ScoringTiles Class

This class displays the actions available for the player.

Attributes:

- **private BufferedImage image :** Image of background photo.

- **private String title :** title.
- **private Font titleFont :** titles's font.
- **private int imagedButtonWidth :** imaged buttons width.
- **private int imagedButtonHeight :** imaged buttons height.
- **private int buttonWidth :** buttons width.
- **private int buttonHeight :** buttons height.

Methods:

- **ActionScreen() :** constructor of actionScreen.
- **render(Graphics2D) :** renders all the graphics.

4.4.3 Model Class Interfaces:

This section is responsible of the players attributes and fractions. It creates objects of bonus, favor, scoring, town tiles by calling their classes. Players have features like power, points, etc. and they are updated by the classes methods like updatePower(), updatePoints(), etc. These methods are called from game manager class. This section also has the fraction class that inherits the fraction types according to players choose.

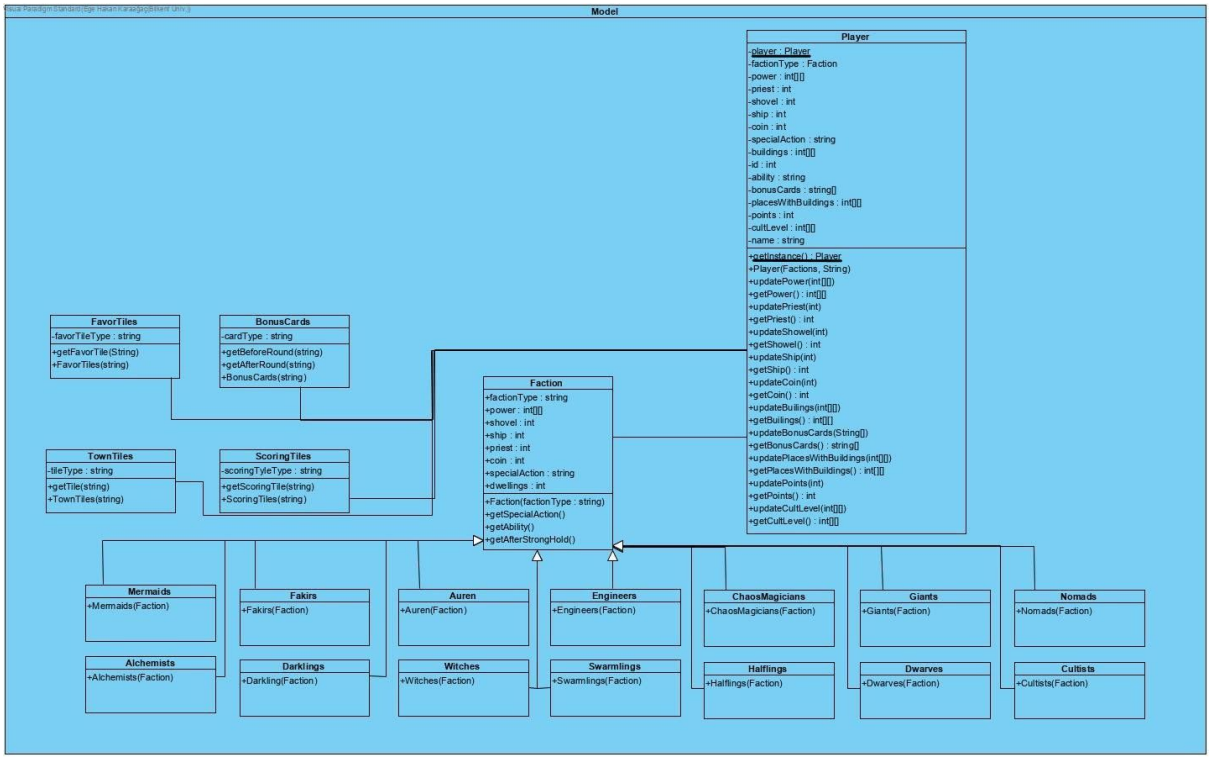

Figure 20 All Game Model Diagram

Faction Class:

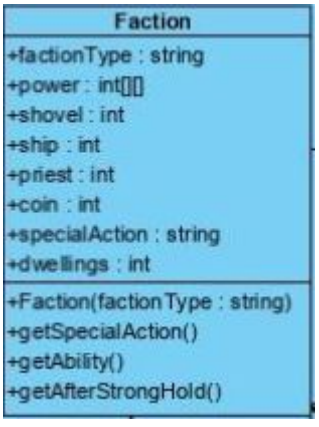

Figure 21: Faction Class

Attributes:

- **private string factionType :** This attribute holds the type of the faction as string.
- **private int [][] power:** This attributes holds the power of the faction.
- **private int shovel:** This attributes holds the number of the shovel of the faction.
- **private int ship:** This attribute holds the number of ships of the faction.
- **private int priest:** This attribute holds the number of priests of the faction.
- **private int coin:** This attribute holds the number of coins of the faction.
- **private string specialAction:** This attribute holds special action of a faction as a string.
- **private int dwellings:** This attribute holds the number of dwellings of the faction.

Constructor:

- **Faction(string):** Constructor for the Faction class.

Methods:

- **public string getSpecialAction():** This method gets the special action of the faction.
- **public string getAbility():** This method gets the ability of the faction.
- **public string getAfterStrongHold:** This method gets the power after strong hold.

Player Class:

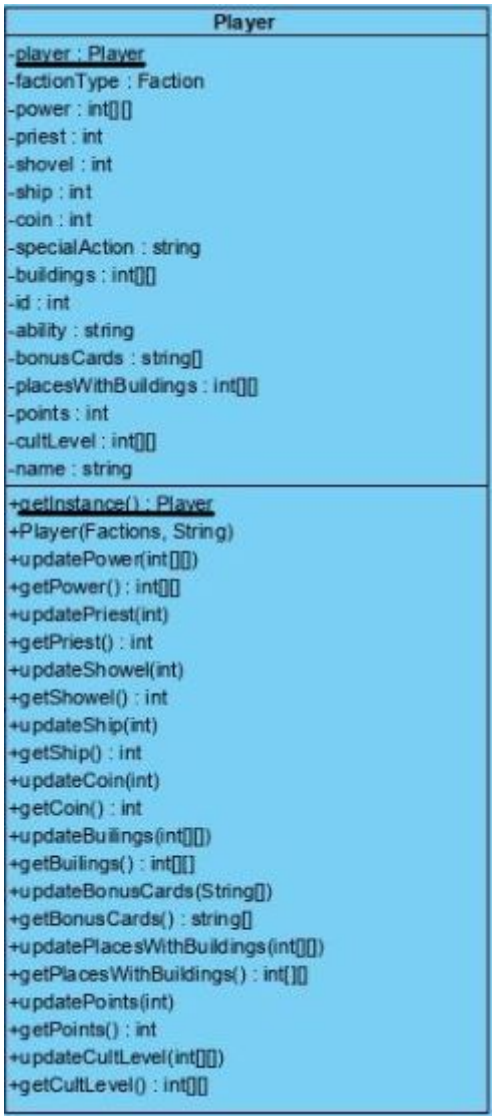

Figure 22: Player Class

Attributes:

- **private player Player:** An instance of Player class
- **private Faction factionType:** This attribute holds the type of the faction as Faction.
- **private int [][] power:** This attribute holds the power of the player as integer array.
- **private int priest:** This attribute holds the number of the priest of the player as integer.
- **private int shovel:** This attribute holds the number of the shovel of the player as integer.
- **private int ship:** This attribute holds the number of ships of the player as integer.
- **private int coin:** This attribute holds the coin number of the player as integer.

- **private string specialAction:** This attribute holds the special action of the player as string.
- **private int [][] buildings:** This attribute holds the building numbers of the player as integer array.
- **private int id:** This attribute holds the id of the player as integer.
- **private string ability:** This attribute holds the ability of the player as string.
- **private string [] bonusCards:** This attribute holds the bonus cards of the player by using array.
- **private int [][] placesWithBuildings:** This attribute holds the numbe rof places with buildings of the player as integer array.
- **private int points:** This attribute holds the points of the player as integer.
- **private int [][] cultLevel:** This attribute holds the cult levels of the player as integer array.
- **private string name:** This attribute holds the name of the player as string.

Constructor:

- **Player(Factions,String):** Constructor for the Player class.

Methods:

- **Player getInstance():** Makes sure there is only one instance of Player. It creates an instance of Player only once and provides the same instance on every function call.
- **public void updatePower(int[][]):** This method updates the power number player.
- **public int[][] getPower():** This method returns the power of the player.
- **public void uptadePriest(int):** This method updates the change of the location of priest.
- **public int getPriest():** This method returns the priest of the player.
- **public void updateShovel(int):** This method updates the change of the location of shovel.
- **public int getShovel():** This method returns the shovel of the player.
- **public void updateShip(int):** This method updates the change of the location of ship.
- **public int getShip():** This method returns the ship of the player.
- **public boolean updateCoin(int):** This method determines the update of the coin number of the player done or not.
- **public void getCoin():** This method returns the coin of the player.

- **public void updateBuildings(int[][]):** This method updates the change of the location of the buildings.
- **public int[][] getBuildings():** This method returns the building of the player.
- **public void updateBonusCard(String[][]):** This method determines the update of the bonus card number of the player done or not.
- **public String[][] getBonusCard():** This method returns the bonusCard of the player.
- **public void updatePlacesWithBuildings(int[][]):** This method determines the update of the places which buildings number of the player done or not.
- **public int[][] getPlacesWithBuildings():** This method returns the placesWithBuildings of the player.
- **public void updatePoints(int):** This method updates points of the player.
- **public int getPoints():** This method returns the points of the player.
- **public void updateCultLevel(int[][]):** This methods determines the update of the cult level of the player done or not.
- **public int[][] getCultLevel():** This method returns the cultLevel of the player.

BonusCards Class:

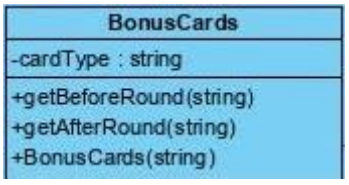

Figure 23:. BonusCards Class

Attributes:

- **private string cardType:** This attribute holds the name of the card type as string.

Constructor:

- **BonusCards(string):** Constructor for the BonusCards class.

Methods:

- **public void getBeforeRound(string):** This method provides player to get the bonus card before the round starts.
- **public void getAfterRound(string):** This method provides player to get the bonus card after the round is end.

ScoringTiles Class:

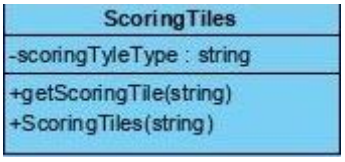

Figure 24: ScoringTiles Class

Attributes:

- **private string scoringTileType:** This attribute holds the name of type of the scoring tile.

Constructor:

- **ScoringTiles(string):** Constructor for the ScoringTiles class.

Methods:

- **public void getScoringTile(string):** This method provides player to get the scoring tile.

TownTiles Class:

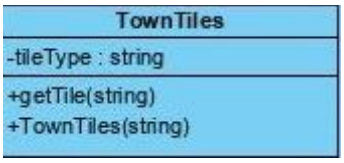

Figure 25:. TownTiles Class

Attributes:

- **private string tileType:** This attribute holds the tile type of the town tile as string.

Constructor:

- **TownTiles(string):** Constructor for the TownTiles class.

Methods:

- **public void getTile(string):** This method provides to get the tile.

FavorTiles Class:

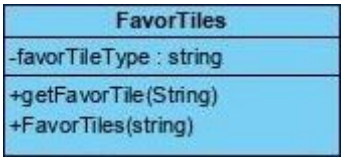

Figure 26:. FavorTiles Class

Attributes:

- **private string favorTileType:** This attribute holds the tile type of the favor tile as string.

Constructor:

- **FavorTiles(string):** Constructor for the FavorTiles class.

Methods:

- **public void getFavorTile(string):** This method provides to get the tile.

Mermaids, Fakirs, Auren, Engineers, ChaosMagicians, Giants, Normaids, Alchemists, Darklings, Witches, Swarmlings, Halflings, Dwarves, Cultists Classes:

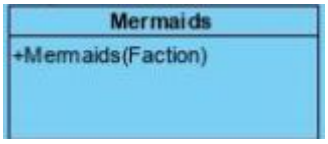

Figure 27: Mermaids Class

Constructor:

- **Mermaids(Faction):** Constructor for the Mermaids class.
- **Fakirs(Faction):** Constructor for the Fakirs class.
- **Auren(Faction):** Constructor for the Auren class.
- **Engineers(Faction):** Constructor for the Engineers class.
- **ChaosMagicians(Faction):** Constructor for the ChaosMagicians class.
- **Giants(Faction):** Constructor for the Giants class.
- **Normaids(Faction):** Constructor for the Normaids class.

- **Alchemists(Faction):** Constructor for the Alchemists class.
- **Darklings(Faction):** Constructor for the Darklings class.
- **Witches(Faction):** Constructor for the Witches class.
- **Swarmlings(Faction):** Constructor for the Swarmlings class.
- **Halflings(Faction):** Constructor for the Halflings class.
- **Dwarves(Faction):** Constructor for the Dwarves class.
- **Cultists(Faction):** Constructor for the Cultists class.