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 of 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 check 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, Javafx

will be used. With the usage of Javafx, 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 and customised 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 Manager, User Interface and Object.

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, a mouse and a keyboard is necessary to play game. Information will provided from the keyboard by users and players will benefit from the mouse to make game actions.

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.

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.
- 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, ther 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 Object

This subsystem contains the game objects. It make its action by the messages of Manager subsystem. When model Object subsystem receives a directive from Manager, it makes alterations and records them. If any updates is necessary on User Interface, Object will detect it and delivers a message to UI subsystem about the alterations which it needs to be done.

3.2 User Interface

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

3.3 Manager

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 Object component by the contents of receiving inputs.

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. As a result of that, memory which the game will allocate may be a bit large.

Portability versus Robustness: For each game, not to break down is so significant feature. If game break down, it will make players angry and even it will result with deletion of the game since players mostly hate to lose the process of game; that's why, we want to make the game that does not break down easily and recover fastly if it fails. Because of these reasons, our priority is robustness instead of portability.

Complexity 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

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.

4.2.2 Observer Design Pattern

Throughout the game process, data of the game will be changed continuously by the moves of players and game needs to keep the track of these data.. Since our software is event driven, we use Observer Design Pattern to observe the alterations on continuously changing data.

4.2.3 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.

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

- **java.util**

This package contains the collections frameworks which will be used in many places as abstract List objects, ImmutableList objects and ObservableList objects.

- **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 Manager Class Interfaces

Our manager interface consists of one general class that is called GameManager Class. 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.

GameManager Class:

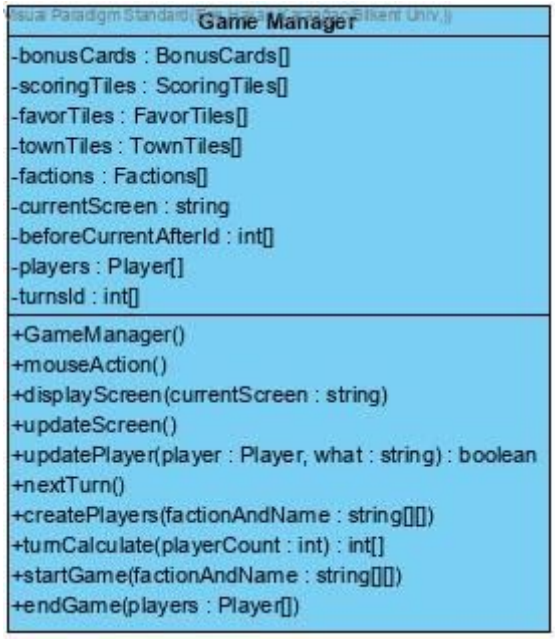

Figure 1. GameManager Class

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 String currentScreen:** This attribute holds the current screen of current game session as string.
- **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.

Methods:

- **public void mouseAction():** This method will be invoked whenever an action occurs with receiving the action events.
- **public void displayScreen(String currentScreen):** This method will be invoked in order to display interface of the game. It gets the current screen as a string.
- **public void updateScreen():** This method will be invoked to update the current screen whenever an action occurs or turn passes one player to another.
- **public boolean updatePlayer(Player player, String what):** This method will be invoked whenever a mouse action occurs and updates player's attributes accoring to action event.
- **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.

4.4.2 User Interface Subsystem

This section of the games class diagram is responsible of each and every gui in the game. It has the menu, how to play, credits, selection, game, gameover screen. 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.

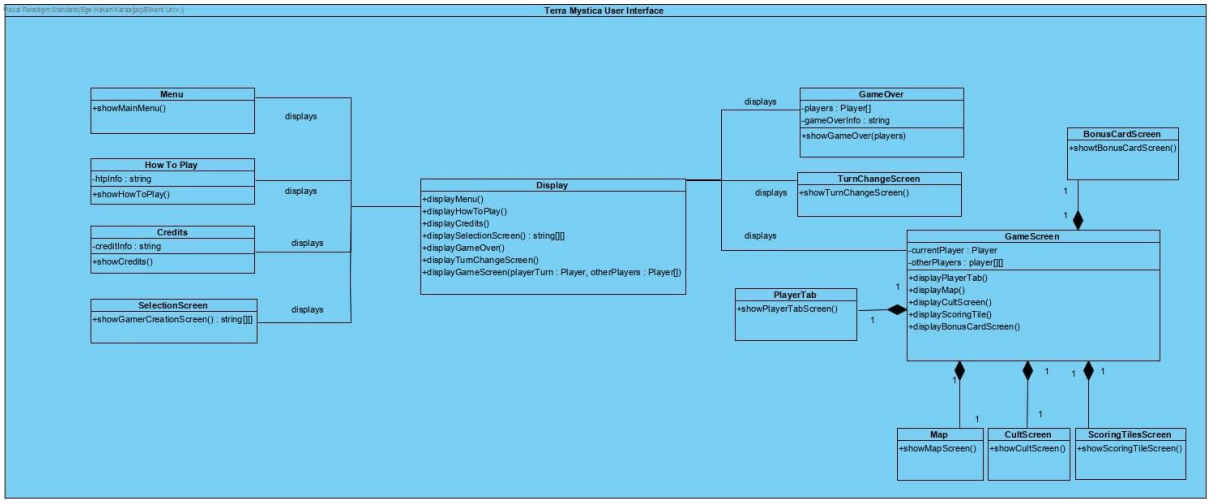

Figure 2. User Interface Diagram

Display Class:

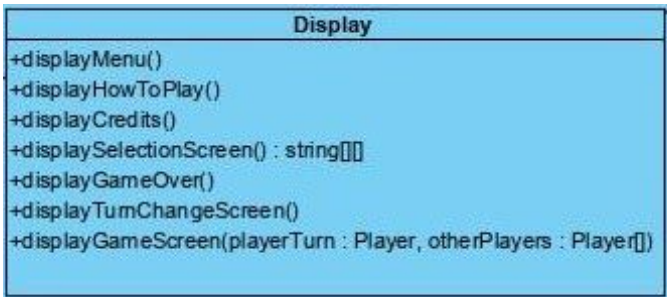

Figure 3. Display Class

This class calls other display classes

Methods:

- **public void displayMenu():** this method calls showMainMenu() method of Menu class
- **public void displayHowToPlay():** this method calls showHowToPlay() method of HowToPlay class

- **public void displayCredits():** this method calls showCredits() method of Credits class.
- **public string displaySelectionScreen():** this method calls showGamerSelectionScreen() method of SelectionScreen class.
- **public void displayGameOver():** this method calls showGameOver() method of GameOver class.
- **public void displayTurnChangeScreen():** this method calls showTurnChangeScreen() method of TurnChangeScreen class.
- **public void displayGameScreen(Player playerTurn, Player[] players):** this method takes current player and other player as parameters and shows gameplay screen by calling methods from GameScreen class.

Menu Class:

Figure 4. Menu Class

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

Methods:

- **public void showMainMenu()**: this method displays the main menu

HowToPlay Class:

Figure 5. HowToPlay Class

This class shows the rules of the game.

Attributes:

- **private final string htpInfo:** this is the text content of game rules.

Methods:

- **public void showHowToPlay()**: this method displays how to play screen.

Credits Class:

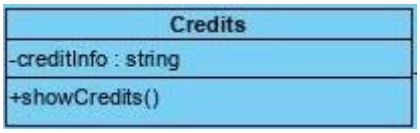

Figure 6. Credits Class

This class shows information about creators of the game.

Attributes:

- **private final string creditInfo:** this is the text content of credits.

Methods:

- **public void showCredits():** this methods displays credits screen

SelectionScreen Class:

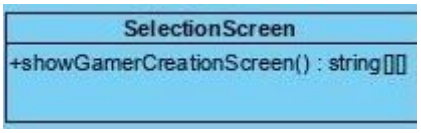

Figure 7. SelectionScreen Class

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

Methods:

- **public string[] showGamerCreationScreen():** this method displays game settings screen before the game starts.

GameOver Class:

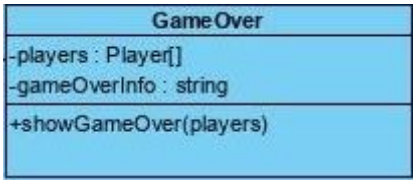

Figure 8. GameOver Class

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

Attributes:

- **private Player[] players**: the list of players
- **private final string gameOverInfo:** the text of game over screen

Methods:

- **public void showGameOverInfo(players):** this method takes the list of all players, sorts and displays them.

TurnChangeScreen Class:

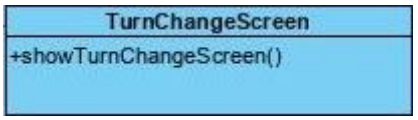

Figure 9. TurnChangeScreen Class

This class displays who's turn is this on each turn change.

Methods:

- **public void showTurnChangeScreen():** this method displays turn change screen.

GameScreen Class:

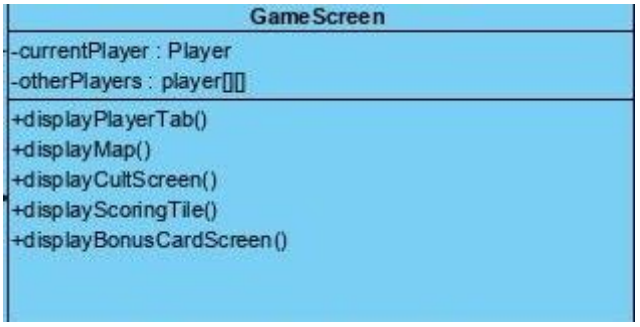

Figure 10. GameScreen Class

This class displays game screen while playing.

Attributes:

- **private Player currentPlayer:** this is the player who is playing at the moment
- **private player[] otherPlayers:** this is list of the other players which are not playing.

Methods:

- **public void displayPlayerTab():** this method displays player tab on game screen by calling showPlayerTabScreen() method of PlayerTab class

- **public void displayMap():** this method displays current game map on game screen by calling showMapScreen() method of Map class.
- **public void displayCultScreen():** this method displays cults on game screen by calling showCultScreen() method of CultScreen class.
- **public void displayScoringTile():** this method displays scoring tile on game screen by calling showScoringTileScreen() method of ScoringTileScreen class.
- **public void displayBonusCardScreen():** this method displays Bonus cards on game screen by calling showBonusCardScreen() method of BonusCardScreen class.

PlayerTab Class:

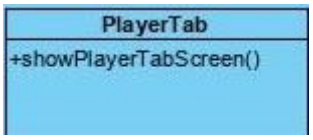

Figure 11. PlayerTab Class

This class displays player tab on game screen.

Methods:

- **public void showPlayerTabScreen():** this method displays player tab.

Map Class:

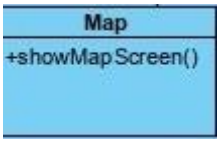

Figure 12. Map Class

This class displays the current status of the map on game screen.

Methods:

- **public void showMapScreen():** this method displays the map.

CultScreen Class:

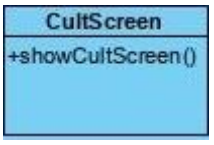

Figure 13. CultScreen Class

This class displays the cults on game screen.

Methods:

- **public void showCultScreen():** this method displays the cults.

ScoringTiles Class:

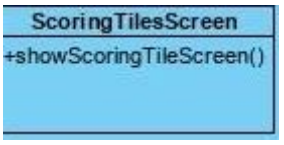

Figure 14. ScringTiles Class

This class displays the scoring tiles on game screen.

Methods:

- **public void showScoringTileScreen():** this method displays the scoring tiles.

BonusCardScreen Class:

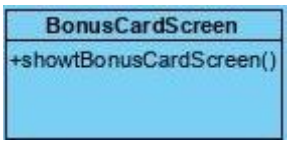

Figure 15. BonusCardScreen Class

This class displays the bonus cards on game screen.

Methods:

- **public void showBonusCardScreen():** this method displays the bonus cards.

4.4.3 Game Object Section:

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.

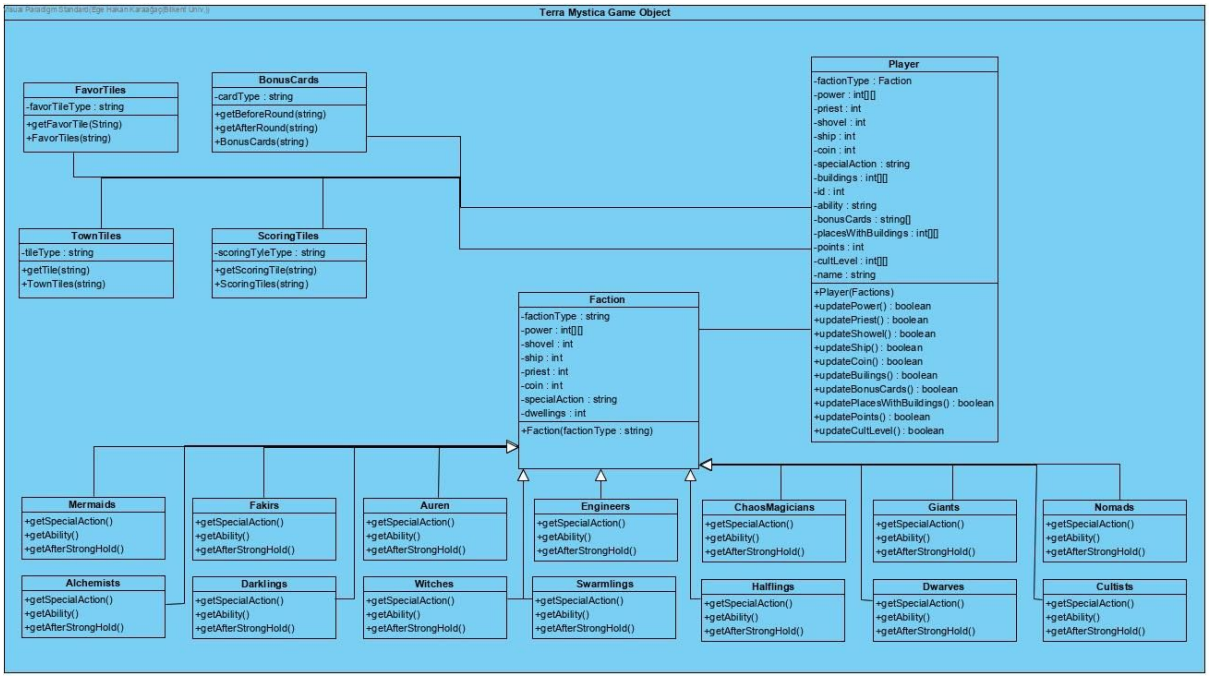

Figure 16. All Game Object Diagram

Faction Class:

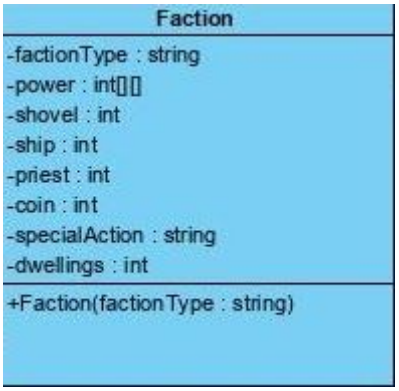

Figure 17. Faction Class

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.

Constructor:

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

Player Class:

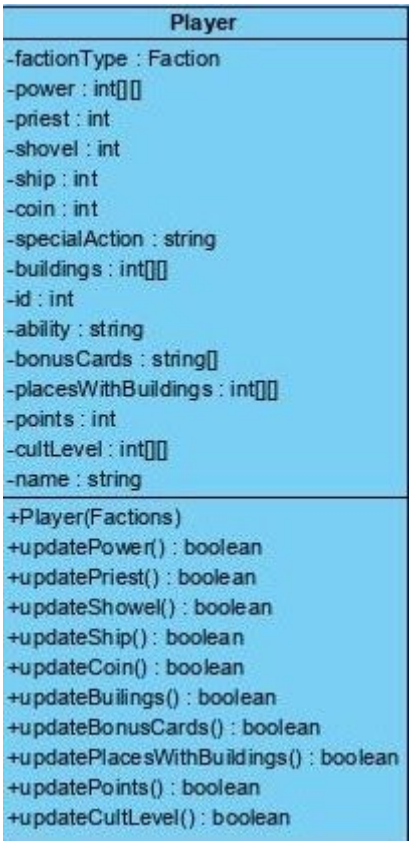

Figure 18. Player Class

Attributes:

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

Methods:

- **public boolean updatePower():** This method updates the power number player.
- **public boolean uptadePriest():** This method updates the change of the location of priest.
- **public boolean updateShovel():** This method updates the change of the location of shovel.
- **public boolean updateShip():** This method updates the change of the location of ship.
- **public boolean updateCoin():** This method determines the update of the coin number of the player done or not.
- **public boolean updateBuildings():** This method updates the change of the location of the buildings.
- **public boolean updateBonusCard():** This method determines the update of the bonus card number of the player done or not.
- **public boolean updatePlacesWithBuildings():** This method determines the update of the places which buildings number of the player done or not.
- **public boolean updatePoints():** This method updates points of the player.
- **public boolean updateCultLevel():** This methods determines the update of the cult level of the player done or not.

BonusCards Class:

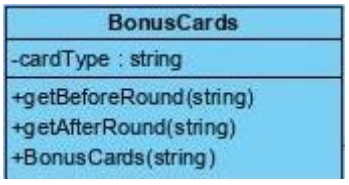

Figure 19. BonusCards Class

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:

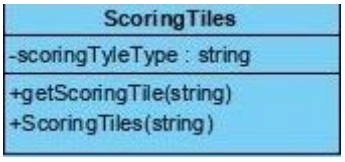

Figure 20. ScoringTiles Class

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:

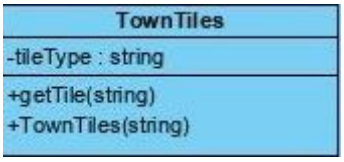

Figure 21. TownTiles Class

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:

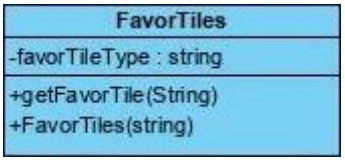

Figure 22. FavorTiles Class

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 Class:

Figure 23. Mermaids Class

Methods:

- **public void getSpecialAction():** This method provides fraction to get the special action.
- **public void getAbility():** This method provides fraction to get the ability.
- **public void getAfterStrongHold():** This method provides fraction get the power after strong hold.

4.4.4 Final Object Design:

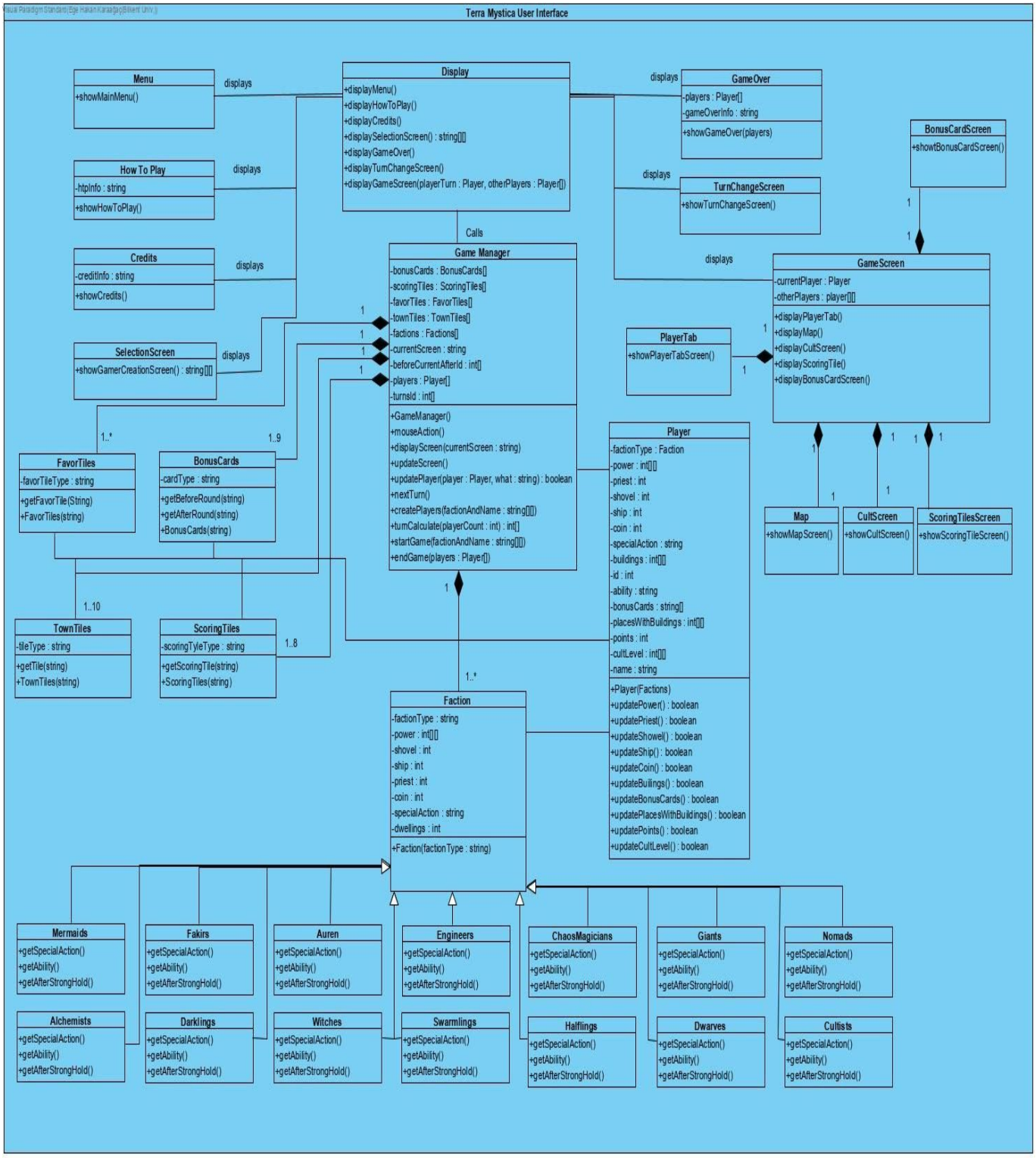

Figure 23. All Class Diagram