Bilkent University

Department of Computer Engineering

Object-Oriented Software Engineering Project

CS 319 1J-TM: Terra Mystica

Design Report

<u>Group Members</u>
Olcay Akman
Zeynep Korkunç
Münevver Uslukılıç
Can Mergenci
İrem Seven
Hüseyin Ata Atasoy

<u>Instructor</u>: Eray Tüzün
<u>Teaching Assistant</u>: Alperen Çetin

Progress Report
April 17, 2020

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the design of the Object-Oriented Software Engineering course CS 319.

Design Report

CS 319 1J-TM: Terra Mystica

Introduction

Purpose of the System

This system is going to be the digital version of the Terra Mystica board game. So, the primary purpose is to create a more efficient, portable and straightforward environment to learn and play the game in comparison to its board version. It eliminates the hassle of learning the rules before playing. Instead, players can learn the rules as they play since the system enforces the rules, thereby leads the players throughout the game. Players also never have to deal with the setup of the game board at the beginning, since the system automatically handles it. Final scoring is calculated in an instance at the end of the game. So, our system is less time consuming and more entertainment focused. Moreover, it is impossible to lose any card or any other piece. There is no need to worry about wear and tear of these pieces. It does not require the physical board and pieces to play. It can be played anywhere that has a computer which has the Java Runtime Environment. In case of lengthy games, players can save their progress, leave the game and continue at any time from where they left off. Thus, it provides flexibility to players. The system offers better user experience than its physical equivalent.

Design Goals

Criteria

Performance Criteria

- Response Time: The player should get a response from the system within a second in order to have a more convenient user experience.
- Throughput: No such criteria exist in the system since this game is turn based.
- Memory: The system should not require more than 500 MB memory space. Competitors require 250 MB memory space. Since we are going to implement the system in Java, it may require more than that.

Dependability Criteria

- Robustness: The system should handle invalid user inputs properly and display appropriate error messages. If the player tries to load a file with a wrong format, the program should not crash, display the cause of the error such as “Wrong File Format”

and prompt for a new file. If the player attempts to make an illegal action during the gameplay, the system should not process the input and nothing happens. Because illegal actions can already be distinguished by their appearance.
- Reliability: The system should behave as expected.
- Availability: Because our system does not run on a server, it should be available at all times it is running.
- Fault Tolerance: The system should automatically save the ongoing game at each state to prevent data loss in case of system crash or power failure.
- Security: No such criteria exist in the system since it is a basic game.
- Safety: The system does not allow playing for long periods of time (2.5 hours). Thus, it avoids endangering the health of the players. After the time limit is reached, the game screen will be frozen for 15 minutes. Then, it will be playable again.

Cost Criteria

- Development Cost: It is equal to the amount of work provided by 6 people for 3 months.
- Deployment Cost: The system does not require installation. It can be run by double clicking the executable file. Game includes the rule book of the original board game. So there is no need to train the users.
- Upgrade Cost: No such criteria exist since there is not a previous system.
- Maintenance Cost: Bug fixes should require at most one week of work of an engineer.
- Administration Cost: No such criteria exist since our system does not require administration.

Maintenance Criteria

- Extensibility: Expansion packs (e.g. Terra Mystica Fire & Ice) may be implemented in the future. So, it should be easy to add new factions, maps, terrain types or different final scoring options to the main game.
- Modifiability: User interface of the system might be changed over time to provide better user experience. So, it should be easily modifiable. Since game rules do not change, there is no need to make it modifiable.
- Adaptability: No such criteria exist since our application domain is fixed.
- Portability: The system runs on every platform that has Java Runtime Environment.
- Readability: The code should be readable to lower the maintenance cost.
- Traceability of requirements: Some parts of the code can easily be mapped to specific requirements (e.g. Save Game, Load Game etc.) whereas some parts cannot (e.g. Non-Functional Requirements).

End User Criteria

- Utility: Game rules are strictly enforced by the system. So, there is nothing to worry about whether some actions are against the rules. The player does not need to know the flow of the game, victory point gains or any other rules. Thus, it provides more utility than its board game counterpart.

- Usability: Navigation and control of the system should be straightforward enough to the extent that players can learn it in 5 minutes. The system also shows the playable actions as another convenience for the players.

Trade-Offs

Fault Tolerance vs. Response Time

The auto-save feature may lower the response time of the system. Because it involves copying game state and writing it to hard-disk. These operations need to be done without interrupting the game flow too much. Our system should prioritize fault tolerance over response time. If response time exceeds one-second, it can be decreased by using concurrency. But, this increases development cost.

Maintenance Cost vs. Development Cost

A system that is easy to maintain has a high development cost in general. Our system should prioritize lowering the maintenance cost as much as possible. Because bugs may disrupt the user experience, they should be handled as soon as possible. Maintainability also paves the way for adding new features to the game. It becomes easier to implement and release new expansion packs of the game. Thus, players have quick access to digital versions of these expansion packs. If development time constraint becomes problematic, our system should try to balance the trade-off between these two criteria.

Performance vs. Portability

Our system has certain advantages over its physical counterpart as we stated earlier. However, if we do not make the system portable enough, some players, who do not have the necessary environment to run it, cannot use it. So, our system should prioritize portability, and thereby will be implemented in Java. Thus, it can be run on everywhere that has the Java Runtime Environment. This may lead to decrease in performance in terms of memory consumption because of the garbage collection technology of Java.

Utility vs. Usability

Our system tries to provide best user experience by eliminating overheads (e.g., requirement of learning the rules to play the game, setting up all components of the game, calculating score of each player, etc.) of its physical counterpart. In addition, utility does not change too much because of the fixed rules of the game. So, our system should prioritize usability over utility. Extra utility should only be added when implementing an expansion pack without impairing usability.

Development Cost vs. Other Criteria

In addition to the criteria above, we have some low priority criteria for the system. For instance, this system should have low response time. It should be robust, safe, extensible and modifiable. Its code should be readable and traceable for some of the requirement specifications. But all of these add up to extra development cost. We have limited time and if time constraint becomes problematic we may need to sacrifice some of these criteria.

High-Level Software Architecture

Subsystem Decomposition

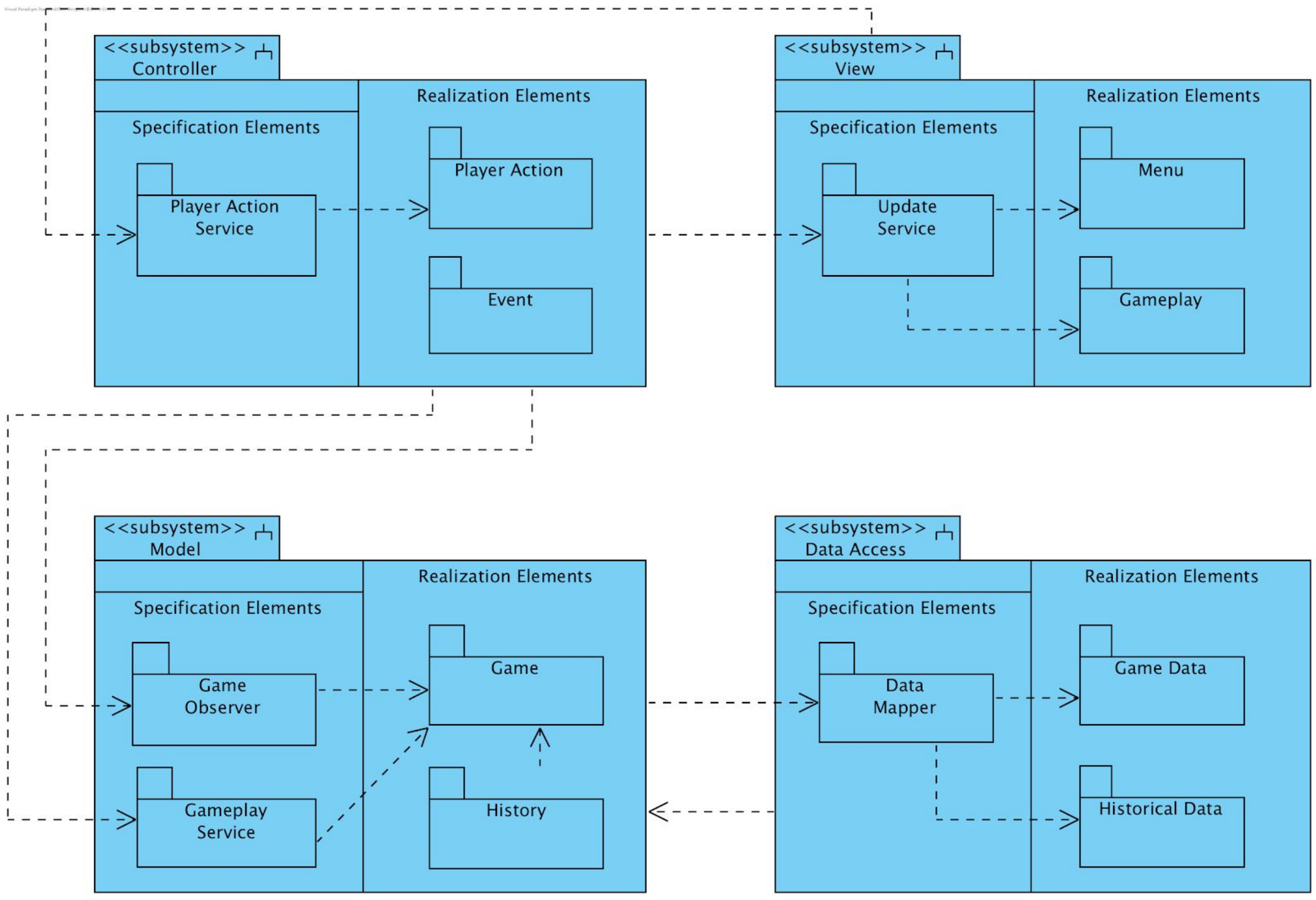


Figure 1: Package Diagram of the System

This system is composed of 4 different subsystems at 2 different layers in “model-view-mediating controller” (MVC) architectural style. “Model” and “Data Access” subsystems are partitions of the bottom layer whereas “Controller” and “View" subsystems are partitions of the top layer. Different layers can easily be implemented by different teams in the group without too much communication burden.

We try to maximize the cohesion by grouping packages with closely related functionalities into the same subsystem. MVC architectural style matches with this need.

We try to minimize the coupling of the system by restricting the dependencies between subsystems. View only depends on the Controller, and has no direct interaction with the Model or Data Access. Controller acts as a mediator in the system and provides communication between Model and View. Model does not know the Controller or View, and only depends on the Data Access for loading a previously saved game.

Since Controller is a mediator in our system, we first considered connecting the Data Access to the Controller to provide communication between Model and Data Access. However, this kind of relation would increase the coupling and might also expose the internal implementation details of the Model to the Controller. So, we have decided not to design in that way. Instead, the Controller interacts with Model to initiate save/load operations, and Model uses Data Access subsystem in turn.

Each subsystem is realized by different packages. Model subsystem is realized by Game and History packages where the former can be mapped to "Play Game” use case and the latter can be mapped to “See History” use case. Data Access subsystem is realized by Game Data and Historical Data packages where the former holds the state of all games and the latter holds some statistical data of overall played games and some historical data of the last played game. Data Access subsystem can be traced back to the “Save Game” and “Load Game” use cases. Controller subsystem is realized by the Event package and Player Action package where the former is responsible for handling the user input and the latter is responsible for ensuring only the legal actions are playable by the players. View subsystem is realized by the Menu and the Gameplay packages where the former is responsible for the menu-related views outside the gameplay, and the latter is responsible for in-game views.

Hardware/Software Mapping

This system consists of a single node. It runs on a single computer and its execution environment is Java Runtime Environment Standard Edition 11. All the objects and subsystems are allocated on the same node. So, there is no need to add a different subsystem for networking.

Off-the-shelf component JavaFX Library is going to be used to implement the Controller and View subsystems instead of building a library on our own for this purpose.

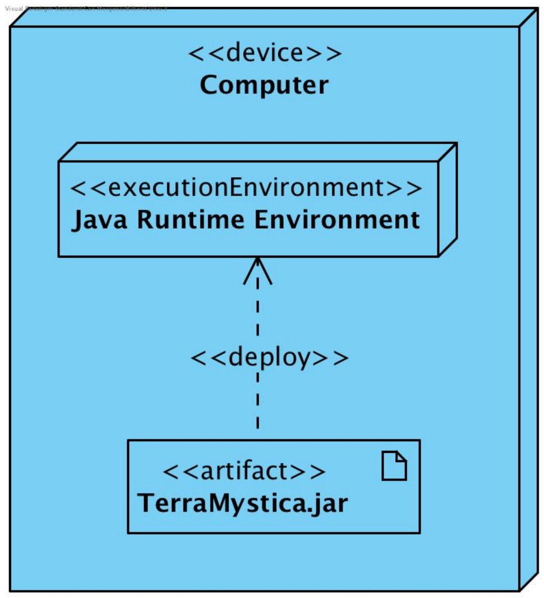


Figure 2: Deployment Diagram of the System

Persistent Data Management

The Data Access subsystem manages the persistent data. Since this system does not perform any queries on its data, we do not need to use a database. Instead, we prefer to store that data in binary flat files which is much simpler to implement.

Only the entity objects, that can be modified by user actions during a game, are stored. If the attributes of an object can be derived or calculated by using stored data related to other objects, then the corresponding object is not stored. Any boundary or control object is not stored for persistence. So, objects related to user preferences like audio and video preferences are not stored as well.

Since the persistent data is stored locally in a flat file and only the attributes of certain entity objects are stored, loading and saving a game may be relatively faster. So, we can reach our low response time design goal in this case.

Access Control & Security

This system is not a multi-user system, that is, it neither creates accounts for different users nor requires a login for authentication. So, there is only one user that launches and plays the game which means that our system has only one actor for all use cases. Different players can only play the game on the same computer. The system does not encrypt any data because security is not a design goal for us.

Players cannot access or change the attributes of other players. Some faction-specific actions are also only accessible for the player who belongs to that faction.

Global Control Flow

The system is a turn-based game. Every action in the gameplay takes place one after the other. There are not many concurrent events. So, general control flow of the final system is going to be event-driven; there is no need to have a separate thread for each event. However, threads may be used to reduce the response time of the system.

We are going to use procedure-driven control to automatically test our subsystems instead of manually invoking events by playing the game.

Boundary Conditions

Configuration

The system does not require installation. The system is created whenever its executable is executed. A game object is instantiated when the player starts a new game with the “Play Game” use case by specifying the number of players and associated factions.

Persistent objects of the system are limited to the entity objects that can be manipulated by the user actions. So, those objects are archived either when the “Player” actor completes the “Save Game” use case or when the system is done with the auto-saving process. Saved games can be loaded back by invoking the “Load Game” use case. So, this is the second way to create a game. But it is created with the predefined attributes in this case.

Game object is destroyed when the “Player” actor is done with the review of the game results at the end of the game and returns to the main menu. Auto-archived Game object is destroyed when the game reaches the termination condition in its steady state (e.g., either when the “Player” completes the “Forfeit Game” use case or the “Player” actor returns to the main menu after reviewing the game results). However, the Game object that is archived intentionally by the “Player” is never destroyed by the system automatically.

Start-up & Shutdown

The system runs on a single node as depicted in its UML deployment diagram. So, it has only one run-time component. It is started upon running its executable file. As soon as the system is initialized and the main menu is displayed, “Player” actor can initiate any use case.

The system is terminated upon exiting the system from the main menu. Once it is shut down, “Player” actors cannot create or modify a game.

Exception Handling

The system automatically saves the ongoing game for every state change in order to be resilient against exceptions. If a game is interrupted due to system crash or power outage, players can continue to play it by loading the auto-saved game.

If auto-saving cannot be done due to a software fault, the system warns the players so that they become aware of the risk of data loss and they can save the game manually.

Players are informed about the types of exceptions, either when the game cannot be saved manually (e.g., insufficient disk space, access denied, etc.), or when the game cannot be loaded (e.g., wrong file format, corrupted file, etc.).

Subsystem Services

Subsystems are specified by services they provide to communicate with other subsystems. The system has 5 main services provided by 4 main subsystems.

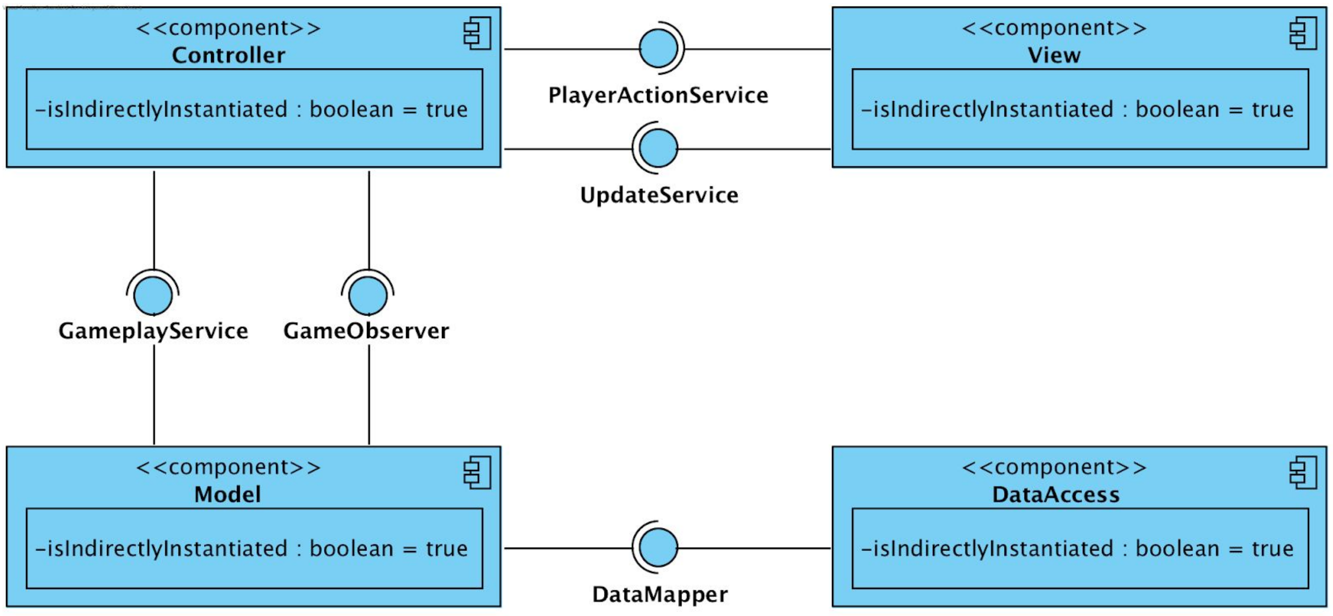

<<component>>
Controller
-isIndirectlyInstantiated : boolean = true
PlayerActionService
UpdateService
<<component>>
View
-isIndirectlyInstantiated : boolean = true
GameplayService
GameObserver
<<component>>
Model
-isIndirectlyInstantiated : boolean = true
DataMapper
<<component>>
DataAccess
-isIndirectlyInstantiated : boolean = true

Figure 3: Component Diagram of the System

Game Observer

Model subsystem provides Game Observer service to the Controller subsystem. It allows the Controller to subscribe to game events, e.g., changes in score of a player, changes in actions available to that player, changes in round, etc. Model notifies the subscribers of that specific event.

Gameplay Service

Model subsystem provides Gameplay service to the Controller subsystem. It updates the game state via player actions. Game is played through this service.

Data Mapper

Data Access subsystem provides Data Mapper service to Model subsystem. Model makes a request to save the current state of the game by using this service. It also allows loading a game by providing the persistent state of the corresponding game.

Player Action Service

Controller subsystem provides Player Action Service to View subsystem. It provides information regarding which player actions are available so that View can be able to enable or disable view of those actions accordingly.

Update Service

View subsystem provides Update Service to Controller Subsystem. It updates the corresponding views according to either user input or changes in the game state.

Low-Level Design

Object Design Trade-Offs

Rapid Development vs Code Navigability

The project uses MVC which provides an implementation process where there is a separation of responsibilities. This results in rapid development because since there is a separation of responsibilities this way different groups can focus on different parts of the project simultaneously. For instance, in our case the development of the view subsystem can simultaneously be done with the model subsystem. However, because of this design pattern we needed to introduce more layers, for instance the model subsystem itself has smaller subsystems within itself and so on. Because of this aspect the navigability between subsystems of this project became harder to maintain, which is a disadvantage.

Number of Options vs Simplicity

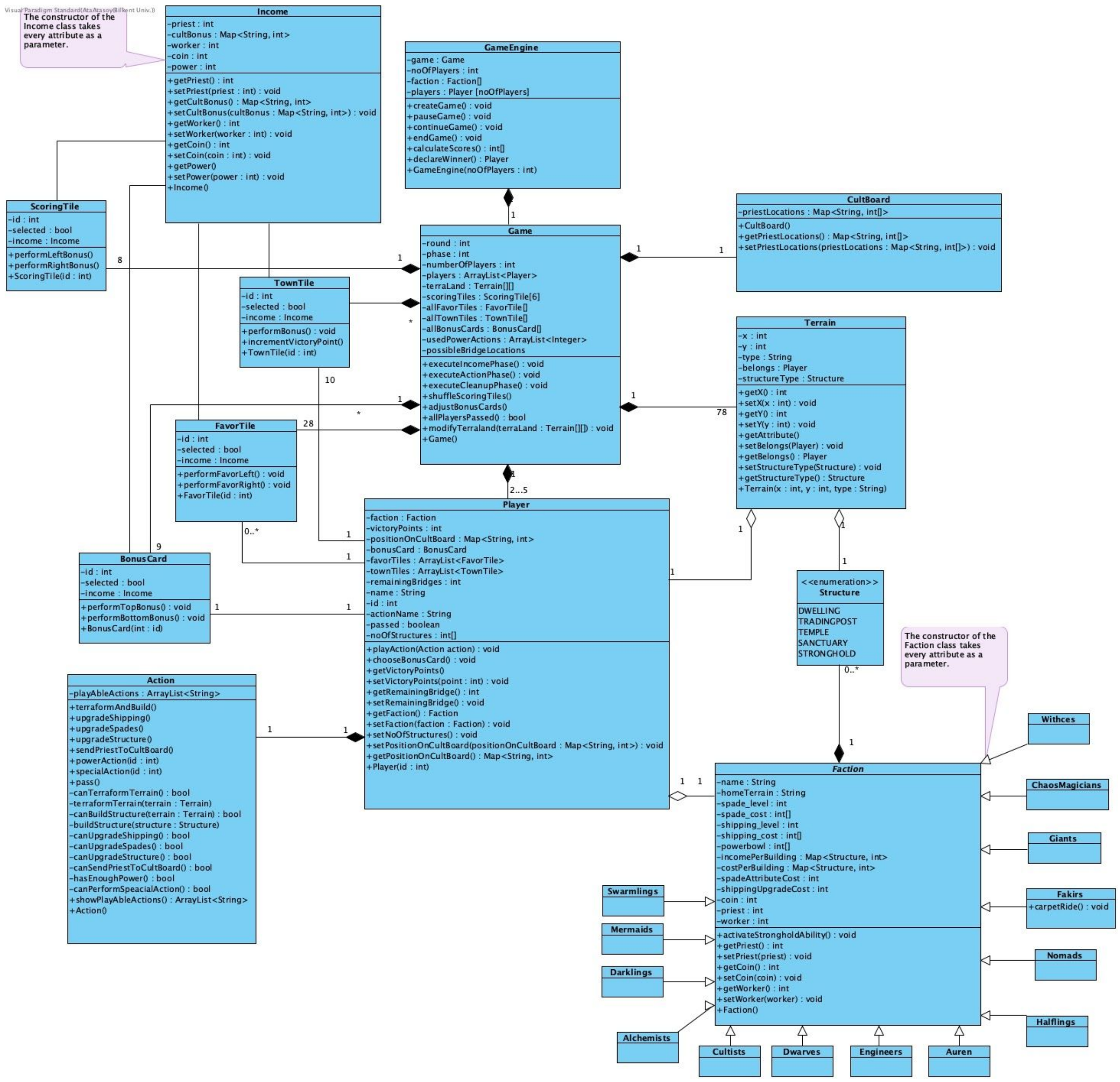
Due to the complex nature of the set of rules of Terra Mystica, we had to include a lot of options during and before the gameplay which is required to increase the compatibility of the digital version of the game with the cardboard version. This is why we used the observer design pattern, which allows a certain object, namely the Game class, to maintain a list of its dependents such as the Player class, the Terrain class, and so on. However, this drastically reduces the simplicity of the game as there are more rules and parameters to keep track of for the player.

Functionality vs Understandability

Terra Mystica is a complex game with several rules to understand. This makes the gameplay harder than many other tabletop games, like Monopoly. During the implementation of the game, in order to increase the understandability we had to compromise functionality, due to the use of the observer design pattern. Adding more pop-up screens to warn the player of the accuracy of their actions during gameplay is one of those compromises however because of this the object design became less functional.

Final Object Design

Model Subsystem



The constructor of the Income class takes every attribute as a parameter.
Income
-priest : int
-cultBonus : Map<String, int>
-worker : int
-coin : int
-power : int
+getPriest() : int
+setPriest(priest : int) : void
+getCultBonus() : Map<String, int>
+setCultBonus(cultBonus : Map<String, int>) : void
+getWorker() : int
+setWorker(worker : int) : void
+getCoin() : int
+setCoin(coin : int) : void
+getPower()
+setPower(power : int) : void
+Income()
GameEngine
-game : Game
-noOfPlayers : int
-faction : Faction[]
-players : Player [noOfPlayers]
+createGame() : void
+pauseGame() : void
+continueGame() : void
+endGame() : void
+calculateScores() : int[]
+declareWinner() : Player
+GameEngine(noOfPlayers : int)
CultBoard
-priestLocations : Map<String, int[]>
+CultBoard()
+getPriestLocations() : Map<String, int[]>
+setPriestLocations(priestLocations : Map<String, int[]>) : void
ScoringTile
-id : int
-selected : bool
-income : Income
+performLeftBonus()
+performRightBonus()
+ScoringTile(id : int)
Game
-round : int
-phase : int
-numberOfPlayers : int
-players : ArrayList<Player>
-terraLand : Terrain[][]
-scoringTiles : ScoringTile[6]
-allFavorTiles : FavorTile[]
-allTownTiles : TownTile[]
-allBonusCards : BonusCard[]
-usedPowerActions : ArrayList<Integer>
-possibleBridgeLocations
+executeIncomePhase() : void
+executeActionPhase() : void
+executeCleanupPhase() : void
+shuffleScoringTiles()
+adjustBonusCards()
+allPlayersPassed() : bool
+modifyTerraland(terraLand : Terrain[][]) : void
+Game()
TownTile
-id : int
-selected : bool
-income : Income
+performBonus() : void
+incrementVictoryPoint()
+TownTile(id : int)
Terrain
-x : int
-y : int
-type : String
-belongs : Player
-structureType : Structure
+getX() : int
+setX(x : int) : void
+getY() : int
+setY(y : int) : void
+getAttribute()
+setBelongs(Player) : void
+getBelongs() : Player
+setStructureType(Structure) : void
+getStructureType() : Structure
+Terrain(x : int, y : int, type : String)
FavorTile
-id : int
-selected : bool
-income : Income
+performFavorLeft() : void
+performFavorRight() : void
+FavorTile(id : int)
Player
-faction : Faction
-victoryPoints : int
-positionOnCultBoard : Map<String, int>
-bonusCard : BonusCard
-favorTiles : ArrayList<FavorTile>
-townTiles : ArrayList<TownTile>
-remainingBridges : int
-name : String
-id : int
-actionName : String
-passed : boolean
-noOfStructures : int[]
+playAction(Action action) : void
+chooseBonusCard() : void
+getVictoryPoints()
+setVictoryPoints(point : int) : void
+getRemainingBridge() : int
+setRemainingBridge() : void
+getFaction() : Faction
+setFaction(faction : Faction) : void
+setNoOfStructures() : void
+setPositionOnCultBoard(positionOnCultBoard : Map<String, int>) : void
+getPositionOnCultBoard() : Map<String, int>
+Player(id : int)
BonusCard
-id : int
-selected : bool
-income : Income
+performTopBonus() : void
+performBottomBonus() : void
+BonusCard(int : id)
<<enumeration>>
Structure
DWELLING
TRADINGPOST
TEMPLE
SANCTUARY
STRONGHOLD
The constructor of the Faction class takes every attribute as a parameter.
Action
-playAbleActions : ArrayList<String>
+terraformAndBuild()
+upgradeShipping()
+upgradeSpades()
+upgradeStructure()
+sendPriestToCultBoard()
+powerAction(id : int)
+specialAction(id : int)
+pass()
-canTerraformTerrain() : bool
-terraformTerrain(terrain : Terrain)
-canBuildStructure(terrain : Terrain) : bool
-buildStructure(structure : Structure)
-canUpgradeShipping() : bool
-canUpgradeSpades() : bool
-canUpgradeStructure() : bool
-canSendPriestToCultBoard() : bool
-hasEnoughPower() : bool
-canPerformSpeacialAction() : bool
+showPlayAbleActions() : ArrayList<String>
+Action()
Faction
-name : String
-homeTerrain : String
-spade_level : int
-spade_cost : int[]
-shipping_level : int
-shipping_cost : int[]
-powerbowl : int[]
-incomePerBuilding : Map<Structure, int>
-costPerBuilding : Map<Structure, int>
-spadeAttributeCost : int
-shippingUpgradeCost : int
-coin : int
-priest : int
-worker : int
+activateStrongholdAbility() : void
+getPriest() : int
+setPriest(priest) : void
+getCoin() : int
+setCoin(coin) : void
+getWorker() : int
+setWorker(worker) : void
+Faction()
Withces
ChaosMagicians
Giants
Fakirs
+carpetRide() : void
Nomads
Halflings
Swarmlings
Mermaids
Darklings
Alchemists
Cultists
Dwarves
Engineers
Auren
1
8
1
1
1
*
10
1
78
28
*
2...5
1
1
1
0..*
1
1
9
1
1
1
0..*
1
1
1
1

Figure 3: Model Subsystem Diagram

The Model subsystem of the game is a result of the MVC design pattern applied to the project as a whole. This subsystem complies with the MVC system design as it functions independent of the view and controller subsystems. However, within the boundaries of itself, the Model subsystem follows along the observer design pattern as in it there are subjects with a list of dependents, called observers. For example the Terrain class is an observer of the Game class, and so on.

GameEngine Class:

This class manages the entire game. It has the following properties and functions:

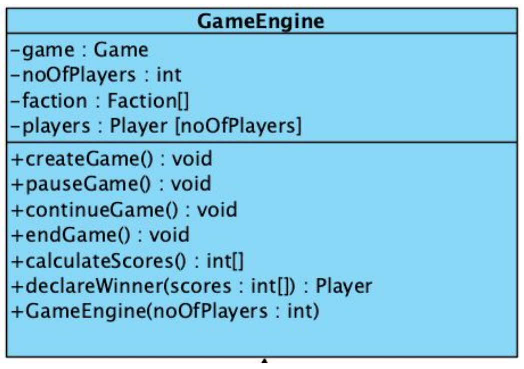


Figure 4: The GameEngine Class

Attributes

- **private Game game :** An instance of the game object.
- **private int noOfPlayers:** Keeps the number of players in the game. Can be 2,3,4 or 5.
- **private Faction[] factions:** An array that stores the different types of Factions in the game.
- **private Player[] players :** Stores the information about the players that are playing the game.

Constructors

- **public GameEngine(int numberOfPlayers)** : Creates a game engine object with a certain number of players that is between two and five.

Methods

- **public void createGame():** Creates a game object.
- **public void pauseGame():** Is executed when a player pauses the game.
- **public void continueGame():** Called to resume the ongoing game.
- **public void endGame():** When the game ends, this function is called.
- **public int[] calculateScores():** Called when the game is over, calculates the scores of each player and returns them in an array.
- **public Player declareWinner(int[] scores):** Takes the final scores of the game as an input and declares the player with the highest score as the winner.

Game Class:

This class represents the game and its components, and is managed by the GameEngine class. It has the following properties and functions:

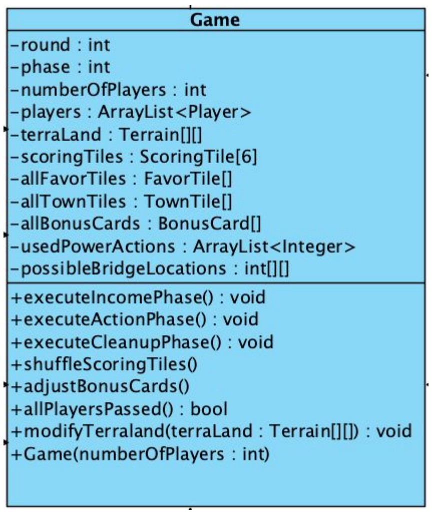

Game
-round : int
-phase : int
-numberOfPlayers : int
-players : ArrayList<Player>
-terraLand : Terrain[][]
-scoringTiles : ScoringTile[6]
-allFavorTiles : FavorTile[]
-allTownTiles : TownTile[]
-allBonusCards : BonusCard[]
-usedPowerActions : ArrayList<Integer>
-possibleBridgeLocations : int[][]
+executeIncomePhase() : void
+executeActionPhase() : void
+executeCleanupPhase() : void
+shuffleScoringTiles()
+adjustBonusCards()
+allPlayersPassed() : bool
+modifyTerraland(terraLand : Terrain[][]) : void
+Game(numberOfPlayers : int)

Figure 5: The Game Class

Attributes

- **private int round:** Keeps track of which round the game is currently in.
- **private int phase:** Keeps track of what the current phase of the current round is.
- **private int numberOfPlayers:** The number of players playing the game.
- **private ArrayList<Player> players:** Keeps the player objects.
- **private Terrain[][] terraLand:** Keeps all the terrains in the TerraLand in a 2D array/matrix.
- **private ScoringTile[6] scoringTiles:** Keeps the scoring tiles of all 6 rounds.
- **private FavorTile[] allFavorTiles:** Keeps the favor tiles stack.
- **private TownTile[] allTownTiles:** Keeps the town tiles stack.
- **private BonusCard[] allBonusCards:** Keeps the bonus cards stack.
- **private ArrayList<Integer> usedPowerActions:** Used to keep track of the power actions that have been used.
- **private int[][] possibleBridgeLocations:** Returns the possible locations to build a bridge onto the map.

Constructors

- **public Game (int numberOfPlayers) :** Creates a game object with a certain amount of players.

Methods

- **public void executeIncomePhase() :** Executed at the beginning of each round to distribute income among players.
- **public void executeActionPhase():** Executed at each round once, after the income phase. Allows the players to perform actions in turns.
- **public void executeIncomePhase():** Executed as the last phase of each round.
- **public void shuffleScoringTiles():** Shuffles the scoring tiles.
- **public void adjustBonusCards():** Adjusts the bonus cards.
- **public bool allPlayersPasses():** Returns a boolean value signaling whether all the players have passed or not.
- **public void modifyTerraland(terraLand: Terrain[][]):** Takes the 2D array of the terraland nodes and their relations to modify it when necessary.

Terrain Class:

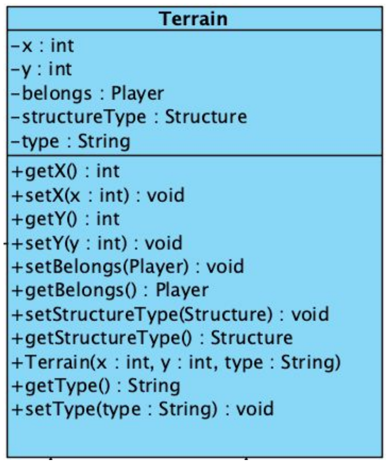

Terrain
-x : int
-y : int
-belongs : Player
-structureType : Structure
-type : String
+getX() : int
+setX(x : int) : void
+getY() : int
+setY(y : int) : void
+setBelongs(Player) : void
+getBelongs() : Player
+setStructureType(Structure) : void
+getStructureType() : Structure
+Terrain(x : int, y : int, type : String)
+getType() : String
+setType(type : String) : void

Figure 6: The Terrain Class

Attributes

- **private int x:** Represents the x coordinate of the Terrain.
- **private int y:** Represents the y coordinate of the Terrain.
- **private Player belongs:** Stores the owner of the Terrain.
- **private String type:** Stores the type of the Terrain, which is one of the following: “Plains”, “Swamp”, “Lakes”, “Forest”, “Mountains”, “Wasteland” or “Desert”.

Constructors

- **public Terrain (int x, int y, String type) :** Cretes the terrain object with a predetermined x-y coordinate and type.

Methods

- **public int getX():** Returns the x coordinate of the terrain.
- **public void setX(int x):** Modifies the x coordinate of the terrain.
- **public int getY():** Returns the y coordinate of the terrain.
- **public void setY(int y):** Modifies the y coordinate of the terrain.
- **public Player getBelongs():** Returns the player, which is the current owner of the Terrain.
- **public void setBelongs(Player p):** Sets the current owner of the Terrain.
- **public Structure getStructureType():** Returns the structure on the Terrain.
- **public void setStructureType(Structure sType):** Sets the structure on the Terrain.
- **public String getType():** Returns the name of the Terrain type.
- **public void setType(String type):** Sets the type of Terrain which can be "Plains", "Swamp", "Lakes", "Forest", "Mountains", "Wasteland" or "Desert".

Cult board Class:

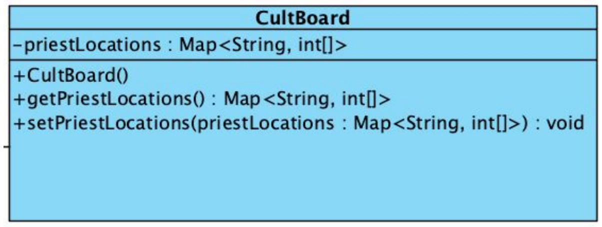

CultBoard
-priestLocations : Map<String, int[]>
+CultBoard()
+getPriestLocations() : Map<String, int[]>
+setPriestLocations(priestLocations : Map<String, int[]>) : void

Figure 7: The Cult Board Class

Attributes

- **private HashMap<String, int[]> :** This hashmap keeps track of the availability of the priest locations for each cult track on the cult board. The keys of this HashMap are "Fire", "Water", "Air" and "Earth". The arrays that it stores keeps track of which player has placed a priest on the cult board. An example case of this HashMap is the following: "priestLocations["Fire"] = [id1, 0, 0, id2]" where id1 and id2 correspond to the id's of the players that have sent to the Fire Cult.

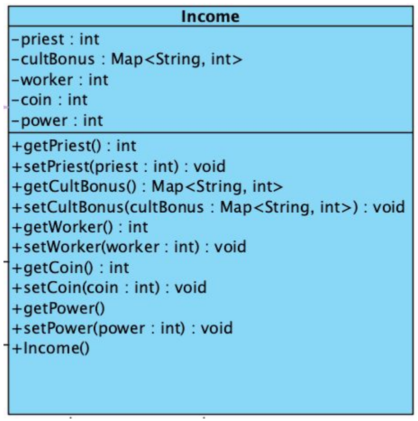
Constructors

- **public CultBoard():** Will create the empty HashMap.

Methods

- **public HashMap<String, int[]> getPriestLocations():** Returns the location of the priests in each cult.
- **public void getPriestLocations(HashMap <String, int[]> h):** Modifies the location of priests in each cult.

Income Class:



Income
-priest : int
-cultBonus : Map<String, int>
-worker : int
-coin : int
-power : int
+getPriest() : int
+setPriest(priest : int) : void
+getCultBonus() : Map<String, int>
+setCultBonus(cultBonus : Map<String, int>) : void
+getWorker() : int
+setWorker(worker : int) : void
+getCoin() : int
+setCoin(coin : int) : void
+getPower()
+setPower(power : int) : void
+Income()

Figure 8: The Income Class

Attributes:

- **private int priest:**
- **private HashMap<String, int> cultBonus:**
- **private int worker:**
- **private int coin:**
- **private int power:**

Constructor:

- **public Income (int priest, HashMap<String, int> cultBonus, int worker, int coin, int power):** Creates an income object that can be gained via a Bonus Card, Favor Tile, Town Tile or a Scoring Tile. Did not list the inputs in the diagram to conserve aesthetics.

Methods:

- **public int getPriest():** Returns the number of priests that will be given as income.
- **public void setPriest(int priest):** Sets the number of priests that will be given as income.

- **public HashMap<String, int> getCultBonus():** Returns the Cult name and the number of positions the player will advance on the corresponding Cult track.
- **public void setCultBonus(HashMap<String, int> h):** Sets the Cult name and the number of positions the player will advance on the corresponding Cult track.
- **public int getWorker():** Returns the number of workers that will be given as income.
- **public void setWorker(int worker):** Sets the number of workers that will be given as income.
- **public int getCoin():** Returns the number of coins that will be given as income.
- **public void setCoin(int coin):** Sets the number of coins that will be given as income.
- **public int getPower():** Returns the amount of power the player will receive.
- **public void setPower(int power):** Sets the amount of power the player will receive.

Favor Tile Class:

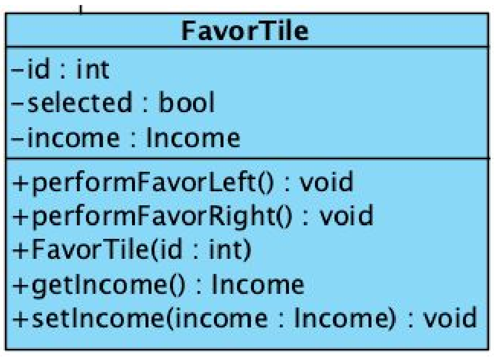


Figure 9: The Favor Tile Class

Attributes

- **private int id:** Stores the id of the Favor Tile. Will be used to distinguish between the different Favor Tiles.
- **private bool selected:** A flag to indicate whether the current Favor Tile has been selected before.
- **private Income :** Indicates the income that a player will gain when they choose the Favor Tile.

Constructors

- **public FavorTile(int id):** Creates the Favor Tile with the corresponding id.

Methods

- **public void performFavorLeft():** Performs the Favor on the left side of the Favor Tile.
- **public void performFavorRight():** Performs the Favor on the right side of the Favor Tile.
- **public Income getIncome():** Returns the income of the Favor Tile.
- **public void setIncome(Income inc):** Sets the income of the Favor Tile.

Scoring Tile class:

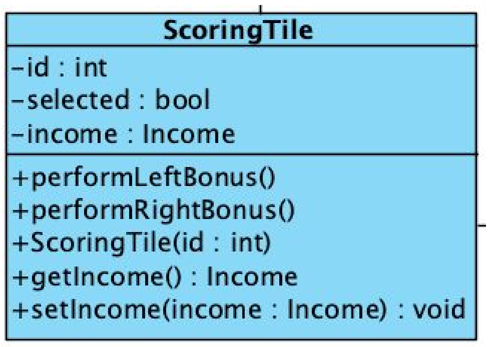


Figure 10: The Scoring Tile Class

Attributes

- **private int id:** Stores the id of the Scoring Tile. Will be used to distinguish between the different Scoring Tiles.
- **private bool selected:** A flag to indicate whether the current Scoring Tile has been selected before.
- **private Income :** Indicates the income that a player will gain when they choose the Scoring Tile.

Constructors

- **public ScoringTile(int id):** Creates the Scoring Tile object with the corresponding id.

Methods

- **public void performLeftBonus():** Performs the bonus on the left side of the Scoring Tile.
- **public void performRightBonus():** Performs the bonu on the right side of the Scoring Tile.
- **public Income getIncome():** Returns the income of the Scoring Tile.
- **public void setIncome(Income inc):** Modifies the income of the Scoring Tile.

Town Tile class:

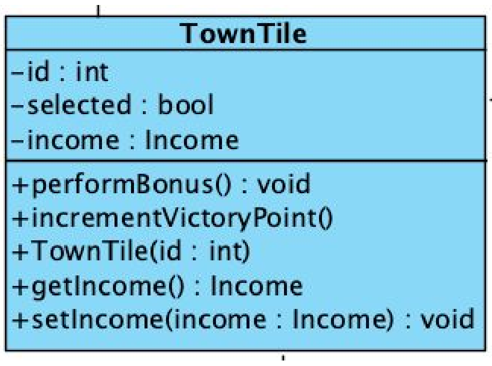


Figure 11: The Town Tile Class

Attributes

- **private int id:** Stores the id of the Town Tile. Will be used to distinguish between the different Town Tiles.
- **private bool selected:** A flag to indicate whether the current Town Tile has been selected before.
- **private Income :** Indicates the income that a player will gain when they choose the Town Tile.

Constructors

- **public TownTile(int id):** Creates the Town Tile object with the corresponding id.

Methods

- **public void incrementVictoryPoint():** Increments the Victory Point of the player.
- **public Income getIncome():** Returns the income of the Town Tile.
- **public void setIncome(Income inc):** Modifies the income of the Town Tile.

Bonus Card class:

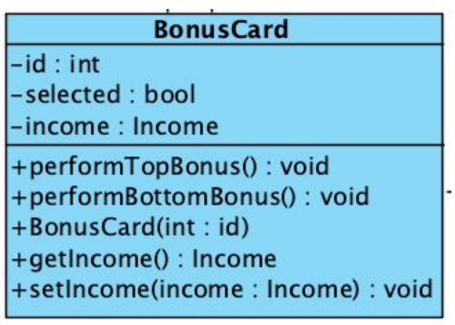


Figure 12: The BonusCard Class

Attributes

- **private int id:** Stores the id of the Town Tile. Will be used to distinguish between the different Bonus Cards.
- **private bool selected:** A flag to indicate whether the current Bonus Card has been selected before.

Constructors

- **public BonusCard (int id):** Creates the Bonus Card object with the corresponding id.

Methods

- **public void performTopBonus():** Performs the bonus described on the top-side of the Bonus Card.
- **public void performBottomBonus():** Performs the bonus described on the bottom-side of the Bonus Card.
- **public Income getIncome():** Returns the income of the Bonus Card.
- **public void setIncome(Income inc):** Modifies the income of the Bonus Card.

Player Class:

Represents a player of the game. It has the following properties and functions:

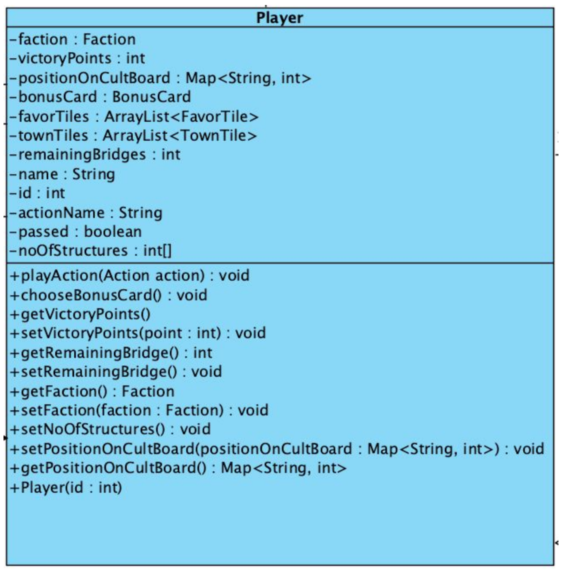

Player
-faction : Faction
-victoryPoints : int
-positionOnCultBoard : Map<String, int>
-bonusCard : BonusCard
-favorTiles : ArrayList<FavorTile>
-townTiles : ArrayList<TownTile>
-remainingBridges : int
-name : String
-id : int
-actionName : String
-passed : boolean
-noOfStructures : int[]
+playAction(Action action) : void
+chooseBonusCard() : void
+getVictoryPoints()
+setVictoryPoints(point : int) : void
+getRemainingBridge() : int
+setRemainingBridge() : void
+getFaction() : Faction
+setFaction(faction : Faction) : void
+setNoOfStructures() : void
+setPositionOnCultBoard(positionOnCultBoard : Map<String, int>) : void
+getPositionOnCultBoard() : Map<String, int>
+Player(id : int)

Figure 13: The Player Class

Attributes

- **private Faction faction:** Holds the faction of the player.
- **private int victoryPoints:** Holds the current victory points of the player.
- **private Map<String, int> positionOnCultBoard:** Keeps the current position of the player on each lane of the cult board.
- **private BonusCard bonusCard:** The bonus card the player currently holds.
- **private ArrayList<FavorTile> favorTiles:** All the favor tiles the player holds.
- **private ArrayList<TownTile> townTiles:**All the town tiles the player holds.
- **private int remainingBridges:** The number of remaining bridges that the player can use.
- **private String name:** The name of the Player.
- **private int id:** The player id.
- **private String actionName:** The last action the player has played.
- **private boolean passed:** A boolean value that holds whether the player has passed or not.
- **private int[] noOfStructures:** Holds the number of each structure the player has.

Constructors

- **public Player(int id):** Creates a player object using the given id as parameter.

Methods

- **public void playAction(Action action):** Allows the player to play the action specified.
- **public void chooseBonusCard():** Lets the player choose a new bonus card.
- **public int getVictoryPoints():** Returns the victory points that the player has.
- **public void setVictoryPoints(int point):** Sets the victory points of the player.
- **public int getRemainingBridge():** Returns the amount of bridges left that the player can use.
- **public void setRemainingBridges(int bridges):** Sets the number of remaining bridges of the player.
- **public Faction getFaction():** Returns the faction of the player.
- **public void setFaction(Faction faction):** Sets the Faction of the player.
- **public Map<String, int> getPositionOnTheCultBoard():** Returns the players current position on the cult board.
- **public void setPositionOnTheCultBoard(Map<String, int> positionOnCultBoard):** Sets the players position on the cult board.

Faction Class:

The Faction superclass as the parent of all factions. There are 14 different factions to be implemented as the child of this class. The superclass has the following properties and functions:

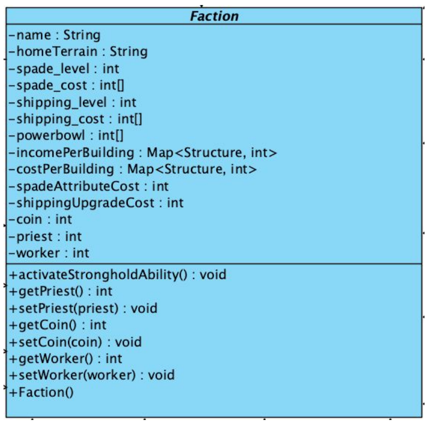


Figure 14: The Faction Class

Attributes

- **private String name:** The name of the Faction.
- **private String homeTerrain:** The home terrain of the faction.
- **private int spadeLevel:** The current spade level of the faction.
- **private int[] spadeCost:** The cost of upgrading spades for that function.

- **private int shippingLevel:** The current shipping level of the player.
- **private int shippingCost:** The cost of upgrading shipping for that faction.
- **private int[] powerBowl:** The amount of power the faction has on each one of the three power bowls.
- **private Map<Structure, int> incomePerBuilding:** The amount of income the faction will earn due to each building they own.
- **private Map<Structure, int> costPerBuilding:** The map of the cost of each building the faction can build/upgrade on the terraland.
- **private int spadeAttributeCost:** The cost of spade attribute.
- **private int shippingUpgradeCost:** The shipping upgrade cost.
- **private int coin:** The amount of coin the faction has.
- **private int priest:** The number of priests the faction has.
- **private int worker:** The number of workers the faction has.

Constructors

- **Public Faction(String name, String homeTerrain, int spadeLevel, int[] spadeCost, int shippingLevel, int shippingCost, int[] powerBowl, Map<Structure, int> incomePerBuilding, Map<Structure, int> costPerBuilding, int spadeAttributeCost, int shippingUpgradeCost, int coin, int priest, int worker):** Creates a Faction object using the given properties as parameter.

Methods

- **public void activateStrongholdAbility():** Allows the faction to activate the Stronghold ability they possess only if they have at least one stronghold built on the terraland.
- **public int getWorker():** Returns the number of workers.
- **public void setWorker(int worker):** Sets the number of workers that the faction has.
- **public int getPriest():** Returns the number of priests.
- **public void setPriest(int priest):** Sets the number of workers that the faction has.
- **public int getCoins():** Returns the number of coins.
- **public void setCoins(int coins):** Sets the number of workers that the faction has.

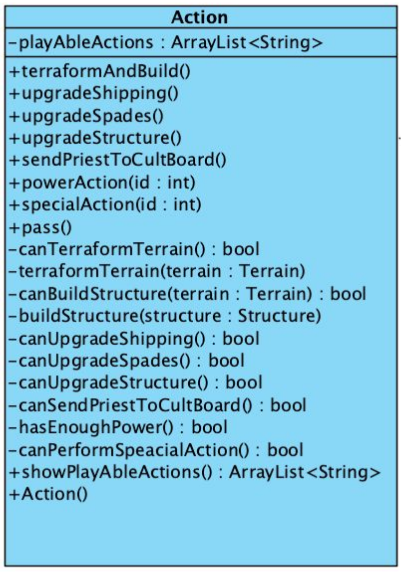
Child Classes

The faction superclass has 14 child classes and each of them are implemented with a common parent of Faction class. Each faction implements the methods and has the properties of the Faction superclass. Since some attributes and methods differ among factions, some of them will be overridden in the child class. These factions are:

- Witches
- ChaosMagicians
- Giants
- Fakirs
- Nomads
- Halflings
- Auren
- Engineers
- Dwarves
- Cultists
- Alchemists
- Darklings
- Mermaids
- Swarmlings

Action Class:

This class represents the actions played by players throughout the game. It has the following properties and functions:



```
Action
-playAbleActions : ArrayList<String>
+terraformAndBuild()
+upgradeShipping()
+upgradeSpades()
+upgradeStructure()
+sendPriestToCultBoard()
+powerAction(id : int)
+specialAction(id : int)
+pass()
-canTerraformTerrain() : bool
-terraformTerrain(terrain : Terrain)
-canBuildStructure(terrain : Terrain) : bool
-buildStructure(structure : Structure)
-canUpgradeShipping() : bool
-canUpgradeSpades() : bool
-canUpgradeStructure() : bool
-canSendPriestToCultBoard() : bool
-hasEnoughPower() : bool
-canPerformSpeacialAction() : bool
+showPlayAbleActions() : ArrayList<String>
+Action()
```

Figure 15: The Action Class

Attributes

- **private ArrayList<String> playableActions:** Keeps the actions that are playable for that round.

Constructors

- **public Action():** Creates an action object.

Methods

- **public void terraformAndBuild():** Allows the player to terraform and/or build (on) a terrain.
- **public void upgradeShipping():** Allows the player to upgrade their shipping level.
- **public void upgradeSpades():** Allows the player to upgrade their spade level.
- **public void upgradeStructure():** Allows the player to upgrade one of their structures' level.
- **public void sendPriestToCultBoard():** Allows the player to send a priest to any lane they prefer on the cult board.
- **public void powerAction(id: int):** Allows the player to play a power action by specifying the power action of choice by giving its id.
- **public void speacialAction(id: int):** Allows the player to play a special action by specifying the special action of choice by giving its id.
- **public void pass():** By using this the player passes their turn for that round.
- **private boolean canTerraformTerrain():** Returns a boolean value saying whether they can terraform terrains or not by checking current coins and workers.
- **private void terraforTerrain(terrain: Terrain):** Terraforms the specified terrain on the Terraland to the hometerrain of the player's faction.
- **private boolean canBuildStructure(terrain: Terrain):** Returns a boolean value specifying whether the player can build a structure on a given terrain.
- **private void buildStructure(structure: Structure, terrain: Terrain):** Allows the player to build the specified building on the specified terrain.
- **private boolean canUpgradeShipping():** Returns a boolean value specifying whether the player can upgrade their shipping depending on their possessions.
- **private boolean canUpgradeSpades():** Returns a boolean value specifying whether the player can upgrade their spade level depending on their possessions.
- **private boolean canUpgradeStructure():** Returns a boolean value specifying whether the player can upgrade their structures depending on their possessions.
- **private boolean canSendPriestToCultBoard():** Returns a boolean value specifying whether the player can send a priest to the cult board depending on their possessions.
- **private boolean hasEnoughPower(action: Action):** Returns a boolean value specifying whether the player has enough power for the action specified.
- **public ArrayList<String> showPlayableActions():** Displays the playable actions by returning a string array list of the action names.

Structure:

Represents the structure enum of the game and it is responsible for holding the necessary information about the structure types of the game. It has the following properties:

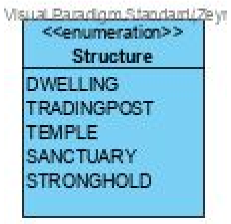
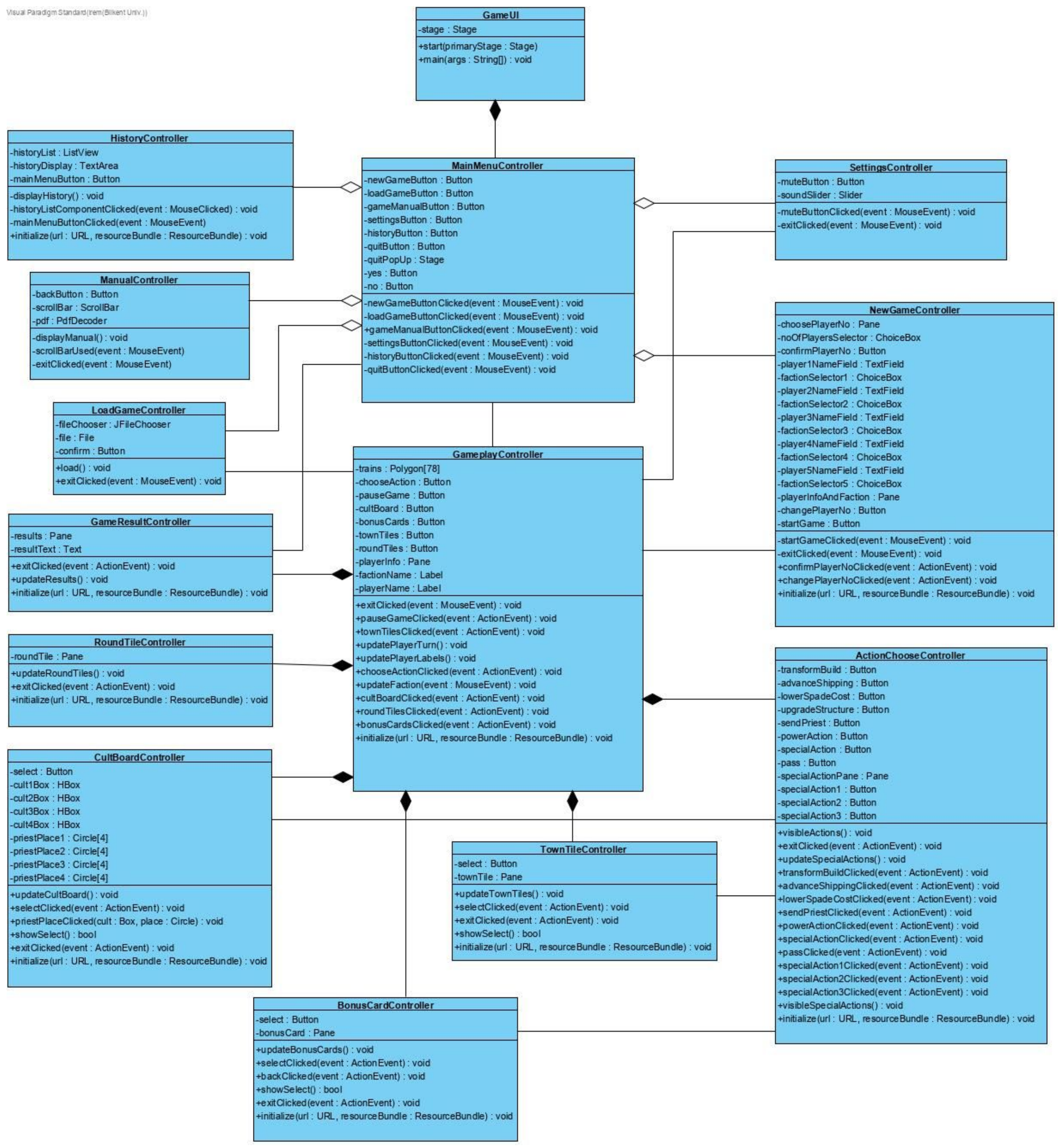


Figure 16: The Structure Enumeration

Description of Enumerations:

- These enums determine the structures that are built by the factions which take place in the terrains.

User Interface Subsystem



Visual Paradigm Standard(irem(Bilkent Univ.))
GameUI
-stage : Stage
+start(primaryStage : Stage)
+main(args : String[]) : void
HistoryController
-historyList : ListView
-historyDisplay : TextArea
-mainMenuButton : Button
-displayHistory() : void
-historyListComponentClicked(event : MouseClicked) : void
-mainMenuButtonClicked(event : MouseEvent)
+initialize(url : URL, resourceBundle : ResourceBundle) : void
MainMenuController
-newGameButton : Button
-loadGameButton : Button
-gameManualButton : Button
-settingsButton : Button
-historyButton : Button
-quitButton : Button
-quitPopUp : Stage
-yes : Button
-no : Button
-newGameButtonClicked(event : MouseEvent) : void
-loadGameButtonClicked(event : MouseEvent) : void
+gameManualButtonClicked(event : MouseEvent) : void
-settingsButtonClicked(event : MouseEvent) : void
-historyButtonClicked(event : MouseEvent) : void
-quitButtonClicked(event : MouseEvent) : void
SettingsController
-muteButton : Button
-soundSlider : Slider
-muteButtonClicked(event : MouseEvent) : void
-exitClicked(event : MouseEvent) : void
ManualController
-backButton : Button
-scrollBar : ScrollBar
-pdf : PdfDecoder
-displayManual() : void
-scrollBarUsed(event : MouseEvent)
-exitClicked(event : MouseEvent)
NewGameController
-choosePlayerNo : Pane
-noOfPlayersSelector : ChoiceBox
-confirmPlayerNo : Button
-player1NameField : TextField
-factionSelector1 : ChoiceBox
-player2NameField : TextField
-factionSelector2 : ChoiceBox
-player3NameField : TextField
-factionSelector3 : ChoiceBox
-player4NameField : TextField
-factionSelector4 : ChoiceBox
-player5NameField : TextField
-factionSelector5 : ChoiceBox
-playerInfoAndFaction : Pane
-changePlayerNo : Button
-startGame : Button
-startGameClicked(event : MouseEvent) : void
-exitClicked(event : MouseEvent) : void
+confirmPlayerNoClicked(event : ActionEvent) : void
+changePlayerNoClicked(event : ActionEvent) : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void
LoadGameController
-fileChooser : JFileChooser
-file : File
-confirm : Button
+load() : void
+exitClicked(event : MouseEvent) : void
GameplayController
-trains : Polygon[78]
-chooseAction : Button
-pauseGame : Button
-cultBoard : Button
-bonusCards : Button
-townTiles : Button
-roundTiles : Button
-playerInfo : Pane
-factionName : Label
-playerName : Label
+exitClicked(event : MouseEvent) : void
+pauseGameClicked(event : ActionEvent) : void
+townTilesClicked(event : ActionEvent) : void
+updatePlayerTurn() : void
+updatePlayerLabels() : void
+chooseActionClicked(event : ActionEvent) : void
+updateFaction(event : MouseEvent) : void
+cultBoardClicked(event : ActionEvent) : void
+roundTilesClicked(event : ActionEvent) : void
+bonusCardsClicked(event : ActionEvent) : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void
GameResultController
-results : Pane
-resultText : Text
+exitClicked(event : ActionEvent) : void
+updateResults() : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void
RoundTileController
-roundTile : Pane
+updateRoundTiles() : void
+exitClicked(event : ActionEvent) : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void
ActionChooseController
-transformBuild : Button
-advanceShipping : Button
-lowerSpadeCost : Button
-upgradeStructure : Button
-sendPriest : Button
-powerAction : Button
-specialAction : Button
-pass : Button
-specialActionPane : Pane
-specialAction1 : Button
-specialAction2 : Button
-specialAction3 : Button
+visibleActions() : void
+exitClicked(event : ActionEvent) : void
+updateSpecialActions() : void
+transformBuildClicked(event : ActionEvent) : void
+advanceShippingClicked(event : ActionEvent) : void
+lowerSpadeCostClicked(event : ActionEvent) : void
+sendPriestClicked(event : ActionEvent) : void
+powerActionClicked(event : ActionEvent) : void
+specialActionClicked(event : ActionEvent) : void
+passClicked(event : ActionEvent) : void
+specialAction1Clicked(event : ActionEvent) : void
+specialAction2Clicked(event : ActionEvent) : void
+specialAction3Clicked(event : ActionEvent) : void
+visibleSpecialActions() : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void
CultBoardController
-select : Button
-cult1Box : HBox
-cult2Box : HBox
-cult3Box : HBox
-cult4Box : HBox
-priestPlace1 : Circle[4]
-priestPlace2 : Circle[4]
-priestPlace3 : Circle[4]
-priestPlace4 : Circle[4]
+updateCultBoard() : void
+selectClicked(event : ActionEvent) : void
+priestPlaceClicked(cult : Box, place : Circle) : void
+showSelect() : bool
+exitClicked(event : ActionEvent) : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void
TownTileController
-select : Button
-townTile : Pane
+updateTownTiles() : void
+selectClicked(event : ActionEvent) : void
+exitClicked(event : ActionEvent) : void
+showSelect() : bool
+initialize(url : URL, resourceBundle : ResourceBundle) : void
BonusCardController
-select : Button
-bonusCard : Pane
+updateBonusCards() : void
+selectClicked(event : ActionEvent) : void
+backClicked(event : ActionEvent) : void
+showSelect() : bool
+exitClicked(event : ActionEvent) : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void

Figure 17: UI Subsystem Diagram

The user interface subsystem is a representation of the Controller subsystem. It complies with the MVC system design as it functions independently of the model subsystem. Each class in this system, except GameUI, specifies a Controller for the corresponding FXML files. The implementation of the given classes will be made by using JavaFX libraries.

The View subsystem consists of FXML and CSS files. Every FXML file has its own Controller class, and they only include the attributes that will be used in that scene, and the methods that will be called from the corresponding Controller class. Thus, it is not needed to be shown in a separate View system class diagram. FXML files mostly created within the help of the SceneBuilder application, and changes are made within the code when it is needed. Since it only provides basic looking user interface view, CSS files are used to make the view look as we wanted.

GameUI Class:

It is the only class that does not have a corresponding FXML file. It starts the user interface application and sets the primary stage, shares its Stage among other Controller classes.

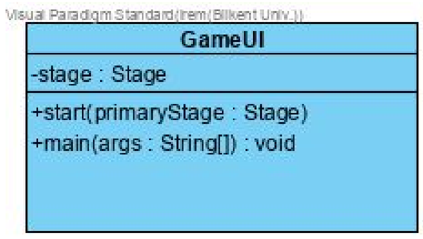

Visual Paradigm Standard(Irem(Bilkent Univ.))
GameUI
-stage : Stage
+start(primaryStage : Stage)
+main(args : String[]) : void

Figure 18: GameUI class

Attributes:

-public static Stage stage: It is used for the primary stage, window. It is made static to be able to access among other classes. That way it will be possible to switch between scenes.

Methods:

-public void start(Stage primaryStage): It starts the user interface application. It sets the main menu as the primary scene, then shows the main menu stage.
-public static void main(String[] args): It will launch the start method.

MainMenuController Class:

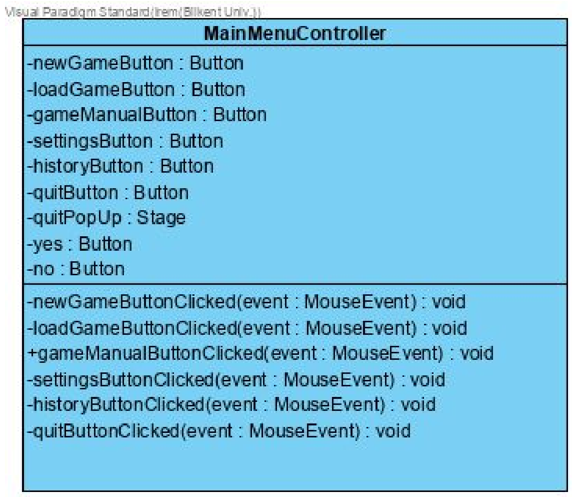

Visual Paradigm Standard(Irem(Bilkent Univ.))
MainMenuController
-newGameButton : Button
-loadGameButton : Button
-gameManualButton : Button
-settingsButton : Button
-historyButton : Button
-quitButton : Button
-quitPopUp : Stage
-yes : Button
-no : Button
-newGameButtonClicked(event : MouseEvent) : void
-loadGameButtonClicked(event : MouseEvent) : void
+gameManualButtonClicked(event : MouseEvent) : void
-settingsButtonClicked(event : MouseEvent) : void
-historyButtonClicked(event : MouseEvent) : void
-quitButtonClicked(event : MouseEvent) : void

Figure 19: MainMenuController class

Attributes:

-private Button newGameButton: Button for opening a new game.
-private Button loadGameButton: Button for loading a previous game.
-private Button gameManualButton: Button for opening the manual.
-private Button historyButton: Button to see history information.
-private Button quitButton: Button to quit game.
-private Stage quitPopUp: Pop up screen for quitting.
-private Button yes: Button for confirming quit.
-private Button no: Button for rejecting quit.

Methods:

-public void newGameButtonClicked(MouseEvent event): Loads newGame.fxml and opens new game window.
-public void loadGameButtonClicked(MouseEvent event): Opens load game window.
-public void settingsButtonClicked(MouseEvent event): Opens settings window.
-public void historyButtonClicked(MouseEvent event): Opens history window.
-public void quitButtonClicked (MouseEvent event): Opens quitPopUp stage which asks the user to confirm quit game operation.

GameplayController Class:

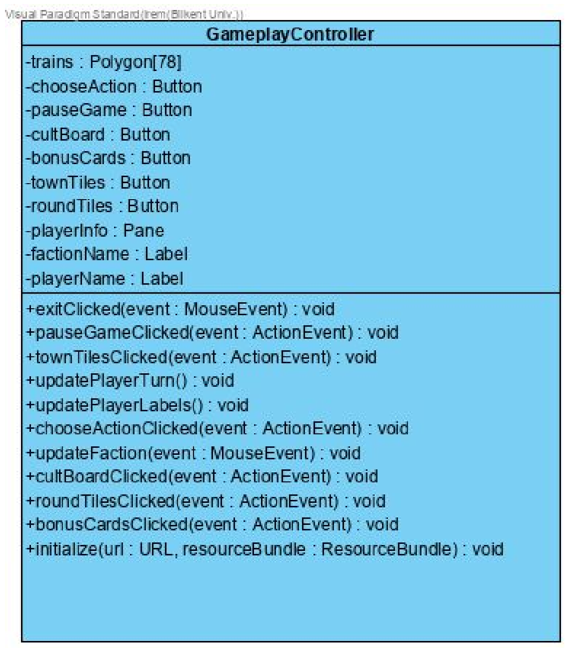

Visual Paradigm Standard(Irem(Bilkent Univ.))
GameplayController
-trains : Polygon[78]
-chooseAction : Button
-pauseGame : Button
-cultBoard : Button
-bonusCards : Button
-townTiles : Button
-roundTiles : Button
-playerInfo : Pane
-factionName : Label
-playerName : Label
+exitClicked(event : MouseEvent) : void
+pauseGameClicked(event : ActionEvent) : void
+townTilesClicked(event : ActionEvent) : void
+updatePlayerTurn() : void
+updatePlayerLabels() : void
+chooseActionClicked(event : ActionEvent) : void
+updateFaction(event : MouseEvent) : void
+cultBoardClicked(event : ActionEvent) : void
+roundTilesClicked(event : ActionEvent) : void
+bonusCardsClicked(event : ActionEvent) : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void

Figure 20: GameplayController class

Attributes:

-private Polygon trains[]: It is for hexagon train spaces. Polygon objects will transformed to shape of hexagons with the help of a css file. There will be 78 polygon objects in this array.
-private Button chooseAction: Button to choose action.
-private Button cultBoard: Button to open cult board stage.
-private Button bonusCards: Button to see bonus cards.
-private Button townTiles: Button to see town tiles.
-private Button roundTiles: Button to see round tiles.
-private Pane playerNoInfo: Area which includes player information. Quantity of it depends on the player number.
-private Label factionName: Faction name for each player pane, area..
-private Label playerName: Player name for each player pane, area.

Methods:

-public void exitClicked(MouseEvent event): Enables to go back to previous stage.
-private void pauseGameClicked(ActionEvent event): Pauses game.
-private townTilesClicked(ActionEvent event): Opens town tile stage without closing main current screen.

-public void updatePlayerTurn(ActionEvent event): Highlights a player pane when its the corresponding player's turn. Made public to be accessed in ActionChooseController class.

-public void updatePlayerLabels(): It updates player information such as power, shipping and spade information in the player's pane. Made public to be accessed in ActionChooseController class.
-private void chooseActionClicked(ActionEvent): It opens action window by loading ActionChoose.fxml to make the user choose an action.
-public void updateFaction(MouseEvent event): It updates a terrain and assigns the information to the player when a polygon clicked on the map. Made public to be accessed in ActionChooseController class.
-private void cultBoardClicked (ActionEvent event): It opens cult board window without closing the current window. Loads cultBoard.fxml.
-private void roundTilesClicked(ActionEvent event): It opens round tile window without closing the current window. Loads roundTile.fxml.
-private void bonusCardsClicked(ActionEvent event): It opens bonus cards window without closing the current window. Loads bonusCards.fxml.
-**private void initialize(URL url, ResourceBundle resourceBundle):** It initializes the game play window, scene. Arranges faction information, player's name in player panes according to the number of players. Also, highlights the first player pane which is going to take action first.

NewGameController Class:

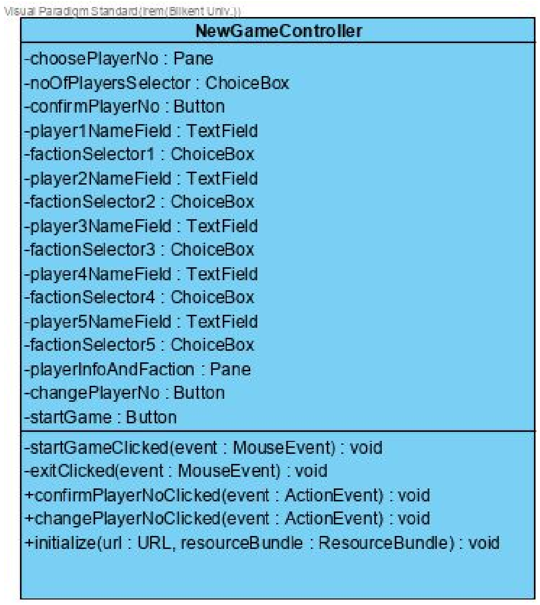

Visual Paradigm Standard(Irem(Bilkent Univ.))
NewGameController
-choosePlayerNo : Pane
-noOfPlayersSelector : ChoiceBox
-confirmPlayerNo : Button
-player1NameField : TextField
-factionSelector1 : ChoiceBox
-player2NameField : TextField
-factionSelector2 : ChoiceBox
-player3NameField : TextField
-factionSelector3 : ChoiceBox
-player4NameField : TextField
-factionSelector4 : ChoiceBox
-player5NameField : TextField
-factionSelector5 : ChoiceBox
-playerInfoAndFaction : Pane
-changePlayerNo : Button
-startGame : Button
-startGameClicked(event : MouseEvent) : void
-exitClicked(event : MouseEvent) : void
+confirmPlayerNoClicked(event : ActionEvent) : void
+changePlayerNoClicked(event : ActionEvent) : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void

Figure 21: NewGameController class

Attributes:

-private Pane choosePlayerNo: The Pane component that holds noOfPlayersSelector, confirmPlayerNo, and all the nameField, factionSelector components for each player.
-private ChoiceBox noOfPlayersSelector: This ChoiceBox object is placed in the choosePlayerNo pane and is used to allow the player to select the number of players; the ChoiceBox has options 2,3,4,5 for the user to select from.
-private Button confirmPlayerNo: This button allows the user to confirm the number of players selected in the ChoiceBox component named noOfPlayersSelector.
-private TextField player1NameField: The input area where player1 inputs their name.
-private ChoiceBox factionSelector1: The ChoiceBox component which allows player1 to select a faction to play.
-private TextField player2NameField: The input area where player2 inputs their name.
-private ChoiceBox factionSelector2: The ChoiceBox component which allows player2 to select a faction to play.
-private TextField player3NameField: The input area where player3 inputs their name.
-private ChoiceBox factionSelector3: The ChoiceBox component which allows player3 to select a faction to play.
-private TextField player4NameField: The input area where player4 inputs their name.
-private ChoiceBox factionSelector4: The ChoiceBox component which allows player4 to select a faction to play.
-private TextField player5NameField: The input area where player5 inputs their name.
-private ChoiceBox factionSelector5: The ChoiceBox component which allows player5 to select a faction to play.

Methods:

-private void startGameClicked(MouseEvent event): Closes the NewGame stage and directs the game to display the Game stage
-private void exitClicked(MouseEvent event): Enables to go back to previous stage.
-private void confirmPlayerNoClicked(MouseEvent event): Sets the playerName input fields, the factionSelectors, and the noOfPlayersSelectors as non-editable.
-private void changePlayerNoClicked(MouseEvent event): Sets the playerName input fields, the factionSelectors, and the noOfPlayersSelectors as editable.
-public void initialize(URL url, ResourceBundle resourceBundle): Initializes the NewGameController view, bounds FXML views with variables and loads the assets.

SettingsController Class:

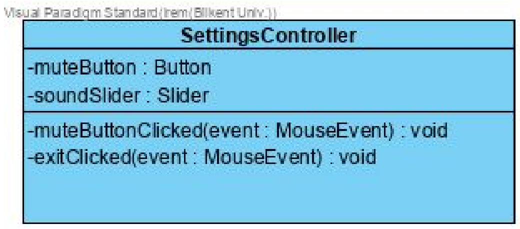

Visual Paradigm Standard(Irem(Bilkent Univ.))
SettingsController
-muteButton : Button
-soundSlider : Slider
-muteButtonClicked(event : MouseEvent) : void
-exitClicked(event : MouseEvent) : void

Figure 22: SettingsController class

Attributes:

-private Button muteButton: Button to mute the sounds in the game.

-private Slider soundSlider: Slider to set the level of the volume.

Methods:

-private void muteButtonClicked(MouseEvent event): Enables to choose if the sounds are off or on.

-public void exitClicked(MouseEvent event): Enables to go back to previous stage.

HistoryController Class:

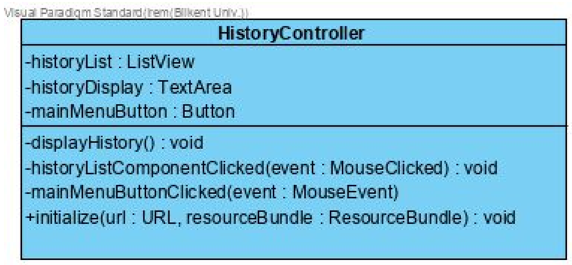

Visual Paradigm Standard(Irem(Bilkent Univ.))
HistoryController
-historyList : ListView
-historyDisplay : TextArea
-mainMenuButton : Button
-displayHistory() : void
-historyListComponentClicked(event : MouseClicked) : void
-mainMenuButtonClicked(event : MouseEvent)
+initialize(url : URL, resourceBundle : ResourceBundle) : void

Figure 23: HistoryController class

Attributes:

-private ListView historyList: List of of the previous games.

-private TextArea historyDisplay: Text area of the history of the games.

Methods:

-private void displayHistory(): Displays the history of the game.

-private void historyListComponentClicked(MouseClicked event): Displays the list of the previous games.

-private mainMenuButtonClicked(MouseEvent event): Returns to the main menu view.

-public void initialize(URL url, ResourceBundle resourceBundle): Initializes the HistoryController view, bounds FXML views with variables and loads the assets.

ManualController Class:

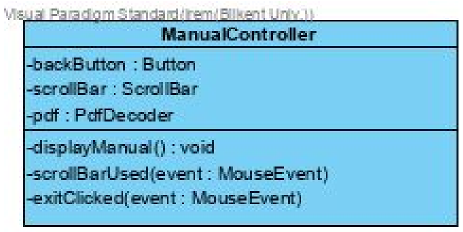

Visual Paradigm Standard(Irem(Bilkent Univ.))
ManualController
-backButton : Button
-scrollBar : ScrollBar
-pdf : PdfDecoder
-displayManual() : void
-scrollBarUsed(event : MouseEvent)
-exitClicked(event : MouseEvent)

Figure 24: ManualController class

Attributes:

-private Button backButton: The Button property to go back to the previous stage display.
-private ScrollBar scrollBar: The ScrollBar component to scroll through the manual displayed in PDF format.
-private PdfDecoder pdf: The PDFDecoder component to display the PDF file that contains the manual to the game.

Methods:

-private void displayManual(): Opens the file that contains the PDF of the game manual and displays the components of the stage.
-private void scrollBarUsed(MouseEvent event): Scrolls through the PDF manual display when the scroll bar is scrolled via the mouse.
-private void exitClicked(MouseEvent event): Enables to go back to previous stage.

LoadGameController Class:

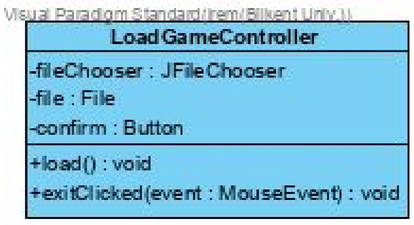

Visual Paradigm Standard(Irem(Bilkent Univ.))
LoadGameController
-fileChooser : JFileChooser
-file : File
-confirm : Button
+load() : void
+exitClicked(event : MouseEvent) : void

Figure 25: LoadGameController class

Attributes:

-private JFileChooser fileChooser: the JFileChooser component to choose the required history files kept in local memory for the game.
-private File file: the File object to hold the chosen file.

-private Button confirm: the button used to confirm the selected choice of history files

Methods:

-public void load(): loads the selected history file. Throws exception if no such file is selected or no such file exists.

-public void exitClicked(MouseEvent event): Enables to go back to previous stage.

ActionChooseController Class:

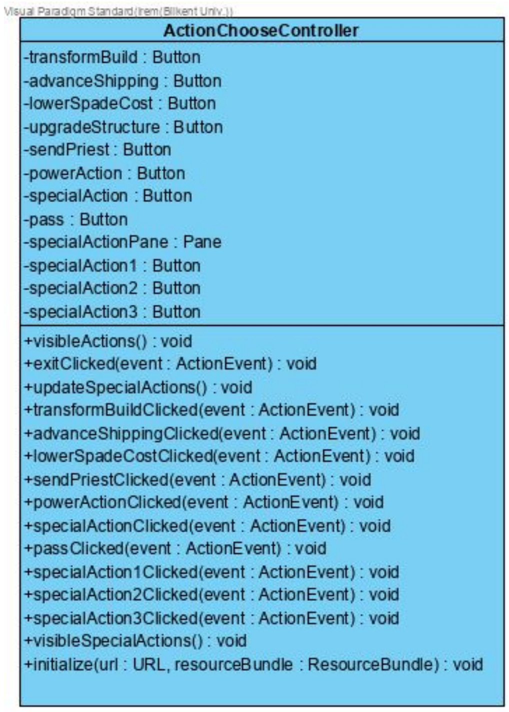

Visual Paradigm Standard(rem(Bilkent Univ.))
ActionChooseController
-transformBuild : Button
-advanceShipping : Button
-lowerSpadeCost : Button
-upgradeStructure : Button
-sendPriest : Button
-powerAction : Button
-specialAction : Button
-pass : Button
-specialActionPane : Pane
-specialAction1 : Button
-specialAction2 : Button
-specialAction3 : Button
+visibleActions() : void
+exitClicked(event : ActionEvent) : void
+updateSpecialActions() : void
+transformBuildClicked(event : ActionEvent) : void
+advanceShippingClicked(event : ActionEvent) : void
+lowerSpadeCostClicked(event : ActionEvent) : void
+sendPriestClicked(event : ActionEvent) : void
+powerActionClicked(event : ActionEvent) : void
+specialActionClicked(event : ActionEvent) : void
+passClicked(event : ActionEvent) : void
+specialAction1Clicked(event : ActionEvent) : void
+specialAction2Clicked(event : ActionEvent) : void
+specialAction3Clicked(event : ActionEvent) : void
+visibleSpecialActions() : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void

Figure 26: ActionChooseController class

Attributes:

-private Button transformBuild: Button to choose transform and build action.
-private Button advanceShipping: Button to choose advance shipping action.
-private Button lowerSpadeCost: Button to choose lower the spade cost action.
-private Button sendPriest: Button to choose send priest to cult action.
-private Button powerAction: Button to perform power action.
-private Button specialAction: Button to perform special action.
-private Button pass: Button to pass as an action.
-private Pane specialActionPane: Shows the special action options. It is only visible right after the specialAciton button is clicked.
-private void specialAction1: Button to select special action one, which is in the specialActionPane.

-private void specialAction2: Button to select special action two, which is in the specialActionPane.
-private void specialAction3: Button to select special action three, which is in the specialActionPane.

Methods:

-private void visibleActions() : It adjusts which action buttons will be visible, playable, in the actions stage, window.
-public void exitClicked(MouseEvent event): Enables to go back to the previous stage.
-private void updateSpecialActions(): Updates the visible special actions that can be performed by the player.Calls updatePlayersLabel function to reflect the change into players information pane.
-private void transformBuildClicked(ActionEvent event): Makes the player to choose a terrain in the game board by opening the Gameplay.fxml.Calls updateFaction function to reflect the change into the map..
-**private void advanceShippingClicked(ActionEvent event):** Advances shipping of a faction. Calls updatePlayersLabel function to reflect the change into players information pane.
-private void lowerSpadeCostClicked(ActionEvent event): Lowers spade cost by one. Calls updatePlayersLabel function to reflect the change into players information pane.
-private void sendPriestClicked(ActionEvent event): Opens cult board window by loading cultBoard.fxml. Also, calls showSelect() to make the selection available in the cult board.
-private void powerActionClicked(ActionEvent event): Creates a temporary stage which asks the user which power action that he/she wants to perform. Then, calls updatePlayersLabel function to reflect the change into players information pane.
-private void specialActionClicked(ActionEvent event): Makes the specialActionPane visible.
-private void passClicked(ActionEvent event): Calls updatePlayersLabel function to reflect the change into players information pane. Also, calls updatePlayerTurn() in GameplayController to make the player pane not to be highlighted again in that round.
-private void specialAction1Clicked(ActionEvent event): Calls corresponding functions for special action one. Also, calls updatePlayersLabel function to reflect the change into players information pane.

-private void specialAction2Clicked(ActionEvent event): Calls corresponding functions for special action two. Also, calls updatePlayersLabel function to reflect the change into players information pane.

-private void specialAction3Clicked(ActionEvent event): Calls corresponding functions for special action three. Also, calls updatePlayersLabel function to reflect the change into players information pane.
-private void visibleSpecialActions() void: Adjusts visible special action buttons in the specialActionPane according to the availability state for that player .
-public void initialize(URL url, ResourceBundle resourceBundle): Makes specialActionsPane invisible, since it will be only visible when the specialActions button is pressed. Initializes the ActionChoose view, bounds FXML views with variables and loads the assets.

TownTileController Class:

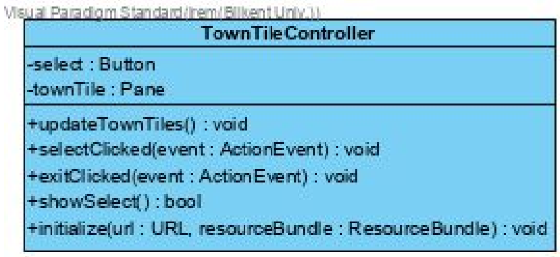

Visual Paradigm Standard(Irem(Bilkent Univ.))
TownTileController
-select : Button
-townTile : Pane
+updateTownTiles() : void
+selectClicked(event : ActionEvent) : void
+exitClicked(event : ActionEvent) : void
+showSelect() : bool
+initialize(url : URL, resourceBundle : ResourceBundle) : void

Figure 27: TownTileController class

Attributes:

-private Button select: Button to select the Town Tile.
-private Pane townTile: Pane that shows all of the available Town Tiles.
Methods:
-public void updateTownTiles(): Updates the available Town Tile after an action happens.
-public void selectClicked(ActionEvent event): Selects the Town Tile.
-public void exitClicked(ActionEvent event): Exits the TownTile pane and returns to the game.
-public bool showSelect(): Updates the current situation(selected / not selected) of the Town Tile.
-public void initialize(URL url, ResourceBundle resourceBundle): Initializes the TownTileController view, bounds FXML views with variables and loads the assets.

BonusCardController Class:

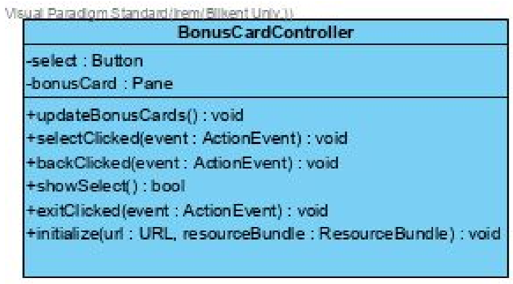

Visual Paradigm Standard(Irem(Bilkent Univ.))
BonusCardController
-select : Button
-bonusCard : Pane
+updateBonusCards() : void
+selectClicked(event : ActionEvent) : void
+backClicked(event : ActionEvent) : void
+showSelect() : bool
+exitClicked(event : ActionEvent) : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void

Figure 28: BonusCardController class

Attributes:

-private Button select: Button component to select the currently chosen Bonus Card.
-private Pane bonusCard: The Pane component to hold the bonus card.

Methods:

-private void updateBonusCards(): Updates the available Bonus Cards after an action happens.
-public void selectClicked(ActionEvent event): Selects the Bonus Card.
-public void exitClicked(ActionEvent event): Exits the Bonus Card pane and returns to the game.
-public bool showSelect(): Updates the current situation(selected / not selected) of the Bonus Card.
-public void initialize(URL url, ResourceBundle resourceBundle): Initializes the BonusCardController view, bounds FXML views with variables and loads the assets.

CultBoardController Class:

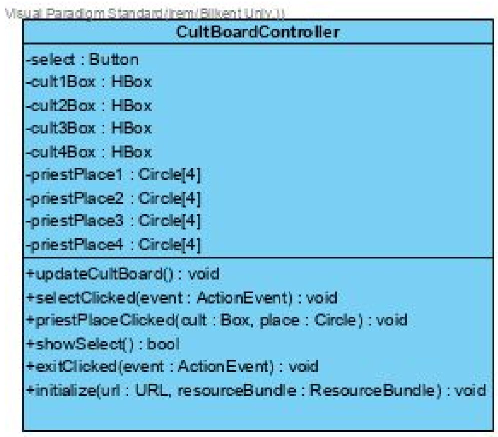

Visual Paradigm Standard(Irem(Bilkent Univ.))
CultBoardController
-select : Button
-cult1Box : HBox
-cult2Box : HBox
-cult3Box : HBox
-cult4Box : HBox
-priestPlace1 : Circle[4]
-priestPlace2 : Circle[4]
-priestPlace3 : Circle[4]
-priestPlace4 : Circle[4]
+updateCultBoard() : void
+selectClicked(event : ActionEvent) : void
+priestPlaceClicked(cult : Box, place : Circle) : void
+showSelect() : bool
+exitClicked(event : ActionEvent) : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void

Figure 29: CultBoardController class

Attributes:

-private Button select: Button for selecting, confirming, to put a priest into specified place.
-private HBox cult1Box: For the first cult's rectangular area that includes cult steps.
-private HBox cult2Box: For the second cult's rectangular area that includes cult steps.
-private HBox cult3Box: For the third cult's rectangular area that includes cult steps.
-private HBox cult4Box: For the fourth cult's rectangular area that includes cult steps.
-private Circle priestPlace1[4]: For the first cult's priest places that include four circular areas.
-private Circle priestPlace2[4]: For the second cult's priest places that include four circular areas.
-private Circle priestPlace3[4]: For the third cult's priest places that include four circular areas.
-private Circle priestPlace4[4]: For the fourth cult's priest places that include four circular areas.

Methods:

-private void updateCultBoard(): Updates cult board view as the players put priest into a place. Corresponding pawns will be shown in the cult steps.
-private void selectClicked(): Makes the necessary changes in the cult board by calling updateCultBoard() function. Also, calls updatePlayersLabel function to reflect the change into players information pane.
-private void priestPlaceClicked(Box cult, Circle place): Specifies the player and the cult place that the player clicked to pass the information into selectClicked().
-public bool showSelect() : It will be false when the player does not select a priest place. This makes the select button invisible.
-**public void exitClicked(ActionEvent event):** Exits the CultBoard window and returns to the previous window.
-public void initialize(URL url, ResourceBundle resourceBundle): Makes select button invisible. Initializes the CultBoardController view, bounds FXML views with variables and loads the assets.

RoundTileController Class:

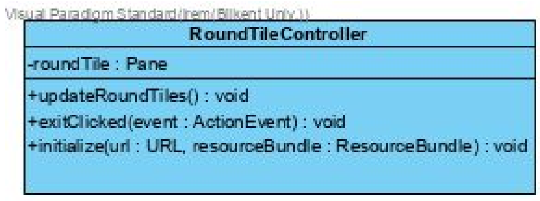

Visual Paradigm Standard(Irem(Bilkent Univ.))
RoundTileController
-roundTile : Pane
+updateRoundTiles() : void
+exitClicked(event : ActionEvent) : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void

Figure 30: RoundTileController class

Attributes:

-private Pane roundTile: Pane that shows all of the Round Tiles in order.

Methods:

-public void updateTownTiles(): Updates the Round Tiles according to the current round.
-public void exitClicked(ActionEvent event): Exits the RoundTile pane and returns to the game.
-public void initialize(URL url, ResourceBundle resourceBundle): Initializes the RoundTileController view, bounds FXML views with variables and loads the assets.

GameResultController Class:

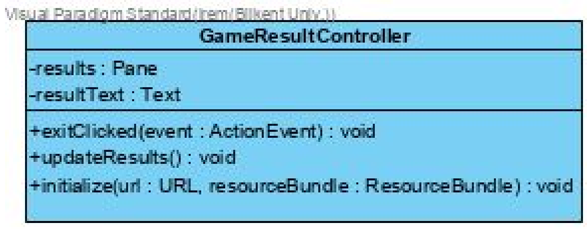

Visual Paradigm Standard(Irem(Bilkent Univ.))
GameResultController
-results : Pane
-resultText : Text
+exitClicked(event : ActionEvent) : void
+updateResults() : void
+initialize(url : URL, resourceBundle : ResourceBundle) : void

Figure 31: GameResultController class

Attributes:

-private Pane results: Pane that shows scores of the players in order, includes resultText.
-private Text resultText: Text shows the winner and the other player's scores.

Methods:

-public void exitClicked(ActionEvent event): Exits the Game Result pane and returns to the main menu.
-public void updateResults(): Updates the scores when game ends.

-public void initialize(URL url, ResourceBundle resourceBundle): Initializes GameResults view, bounds FXML views with variables and loads the assets.

Packages

In this implementation, different packages are defined in order to use in the proper subsystem. Our system possesses four different subsystems and as mentioned before, and the packages above belong to the corresponding subsystems. In addition to the packages introduced by us, additional packages used which belong to the external libraries of java.

Packages Developed

Game:

This package belongs to the Model Subsystem. It includes classes which are responsible for the logic of the game and provide the necessary objects which possess the methods that manage the flow of the game.

History:

It is the second package belongs to the Model Subsystem and it is responsible for displaying the historical data of the past games, e.g. players, their associated factions and victory points at the end of the game. Overall statistics of all games like total games played, total games finished, average victory points earned per player per game is also available. It includes the necessary classes that obtain information from the data access subsystem.

GameData:

This package belongs to the Data Access Subsystem. It included the necessary classes and methods to save the current data of the game in order to prevent any data loss if a system crash occurs.

Historical Data:

This package belongs to the Data Access Subsystem. It possesses the classes which are responsible for the persistence of the history data.

Event:

This package belongs to the Controller Subsystem. It is responsible for handling the user input.

Player Action:

This package belongs to the Controller Subsystem. It includes the classes which are responsible for ensuring only the legal actions are playable by the players.

Menu:

This package belongs to the View Subsystem and it contains the necessary classes and methods responsible for displaying the game menu.

Gameplay:

Gameplay package belongs to the View Subsystem. This package includes the classes and methods that are responsible from the game related views.

External Packages

javafx.application: This package helps to create a stage and scene and makes the scene visible.

javafx.animation : This package provides high level constructs for creating effects and helps to do the transitions based on animations in the game. It also creates timer so that transitions can be regulated in a determined duration.

javafx.css: This package provides applying the styles to the properties by using Cascading Style Sheets.

javafx.event: This package provides the delivery and handling of the events in the application.

javafx.fxml: This package is used for loading objects and initializing the interfaces.

javafx.scene: This package provides the base classes for JavaFX. It fills the background and specify the width and height of the scene. It helps for coordinating the system and transformations. Since our game is based on shapes and their locations, this package is used.

javafx.scene.control: The package is used for being manipulated by user. It provides of the objects like buttons, text fields, text areas, toolbars, scrollbars, menu and etc. so that it helps to control the interactions.

javafx.scene.image: This package provides loading and displaying images. The size, quality and options of the images can be changed.

javafx.input: This package provides handling the events of mouse and keyboard.

javafx.layout: This package is used for managing the interface layout. Pane and background property classes are provided.

javafx.scene.media: This package provides adding audio and video to the application. It is used for adding sounds to the game.

javafx.scene.shape: This package is used for creating two-dimensional shape and performing operations related to them.

javafx.scene.text: This package provides displaying and rendering fonts and texts.

javafx.stage: This package provides the top-level classes for JavaFX. It is going to be used for old game to be loaded and pop up windows to be viewed.

References

Bruegge, B., & Dutoit, A. H. (2014). *Object-oriented software engineering using UML, Patterns, and Java* (3rd ed.). Pearson.

Visual Paradigm. (2020). *Visual Paradigm Standard* (Version 16.1). Visual Paradigm International. https://www.visual-paradigm.com/editions/standard/

Ostertag, H., & Drögemüller J. (n.d.). *Terra Mystica Rule Book.* Feuerland Spiele. http://www.feuerland-spiele.de/dateien/Terra_Mystica_EN_1.0_.pdf

Oracle. (2015, February 10). Retrieved from https://docs.oracle.com/javase/8/javafx/api/overview-summary.html