Bilkent University
Department of Computer Engineering

CS 319 - Object-Oriented Software Engineering
Project Design Report
Iteration 2

Settlers of Catan

Group 1D

Berke Oğuz
Alkım Önen
Kaan Tapucu
İbrahim Eren Tilla
Hasan Yıldırım

Table of Contents

Kaan Alkım Berke Hasan Eren

1. Introduction

1.1 Purpose of the system

1.Introduction

1.1 Purpose of system

Catan is a multiplayer strategy board game. Game is created in such a way that the players could get the conceivable greatest fulfillment from the game. Catan game is a convenient, easy to understand and challenging game which means to engage the clients by including them into the board game. The player's objective is to defeat all the opponents by accessing 10 points. Game has to be sufficient quick, easy to use and with high frame per second for players to play without problem.

1.1.1 Trade-Offs

1. **Development Time vs Performance**

   A Large portion of individuals prefer that Java has the best features because of large community that can help each other and flexibility of java. Furthermore, unlike the C++, we did not need to fix memory leaks that will reduce the development time thanks to the garbage collector.

2. **Functionality vs Usability**

   Catan is a game that has basic controls. Since we made a game that can be played by mostly mouse, it will not have complicated controls. Although catan can be difficult for the players who are playing for the first time, we tried to make our game user-friendly as much as possible. Therefore, we provided a how to play menu that provides users better understanding.

3. **Understandability vs Functionality**

   As it was referenced previously, our game is relied upon to be anything but difficult to learn and justifiable game. In regard to this, we needed to reduce having many different features and usefulness of the game by wiping out disrupting

and complex game materials. As a result, the players would not try to understand, yet would figure out the straightforward reasonable game.

4. Memory vs Maintainability

During the design and the analysis of the game, we tried to utilize oppositiness. Our main objective in this plan was to keep up the game functionalities effectively as could be expected under the circumstances. In any case, by doing that we some parts will have a few methods and properties which they do not need. This pointless traits will cause memory allocation more that the game requirements. Along with these, the attributes that we have, might cause memory allocation. Then again, since we preoccupied the basic qualities however much as could be expected, it would be very simple to keep up the game which will bring about better game experience. By profiting these regular properties of subclasses, players will control game with effectively.

1.1.2 Criteria

1.1.2.1 Usability

Catan is relied upon to be an engaging game for the players which will give them an easy to use interface, with the goal that it was simple for the players to use with focusing on the game itself, instead of understanding the game rules and mechanics. The game framework will use basic strategy game controls, which is the simple use for the players. For our assumption, the vast majority of players avoid the "How to Play" option in the game, pretty much in every gameplay. In any case, since our framework is made so that, it was simple for the players to comprehend it even without the assistance of how to play screen. From the player's perspective, it will be exceptionally simple to keep track of how the turns is finished, when the cities, roads or towns are built, trades, gathering resources and so on.

1.1.2.2 Performance

Execution is one of the significant structure objectives, which is required for the games. We will be using JavaFX library for the better performance.

1.1.2.3 Extendibility

The plan of the game will be able to alter and change its functionalities later on relying upon the client input and feedback. For instance, we can broaden its difficulties by including the time range of each turn, or the quality of resource hexes, cities, roads and so on.

1.1.2.4 Modifiability

Catan game is structured for playing multiplayer. It is simple to adjust the framework with our design structure.

1.1.2.5 Reusability

Without any adjustments, subsystems can be used in different games. They were planned autonomous from the framework.

1.1.2.6 Portability

We chose to actualize in Java, since JavaFX let us implement our game into video game console or personal computer. Therefore, our game can be easily converted to run on different platforms.

1.1.2.7 Performance Criteria

It is significant for the game to have the quick responses for the players' actions. Catan game is planned so its responses were practically rapid, during the game itself, in a show of the liveliness and impacts.

1.2 Design goals

In order to make the framework, design has another significant advance. It recognizes the plan objectives for the framework that we should concentrate on. As it was referenced in our analysis report, there are numerous non-functional requirements of the framework that should be better explained in the design part. At the end of the day, they are the object of the concentration in this report. Following areas are the description of the significant design objectives.

2. High Level Software Architecture

2.1. Subsystem Decomposition

The main aim of the subsystem decomposition is to make the general complexity of the system be decomposed into subsystems that consist of classes that have a strong relationship so that it makes it easier to comprehend and implement later on. The software of our game is decomposed into three main subsystems that have minimal dependency on each other. We have decided to use the MVC (Model-View-Controller) pattern as our main decomposition technique. We connected these 3 main components in a way that would make sure that they are separate enough to maintain separately (coupling), yet connected enough to keep the general relationships between each other (cohesion). The design goals of our project was the key consideration in the design of the subsystem decomposition. The visualization of our subsystem decomposition is as follows:

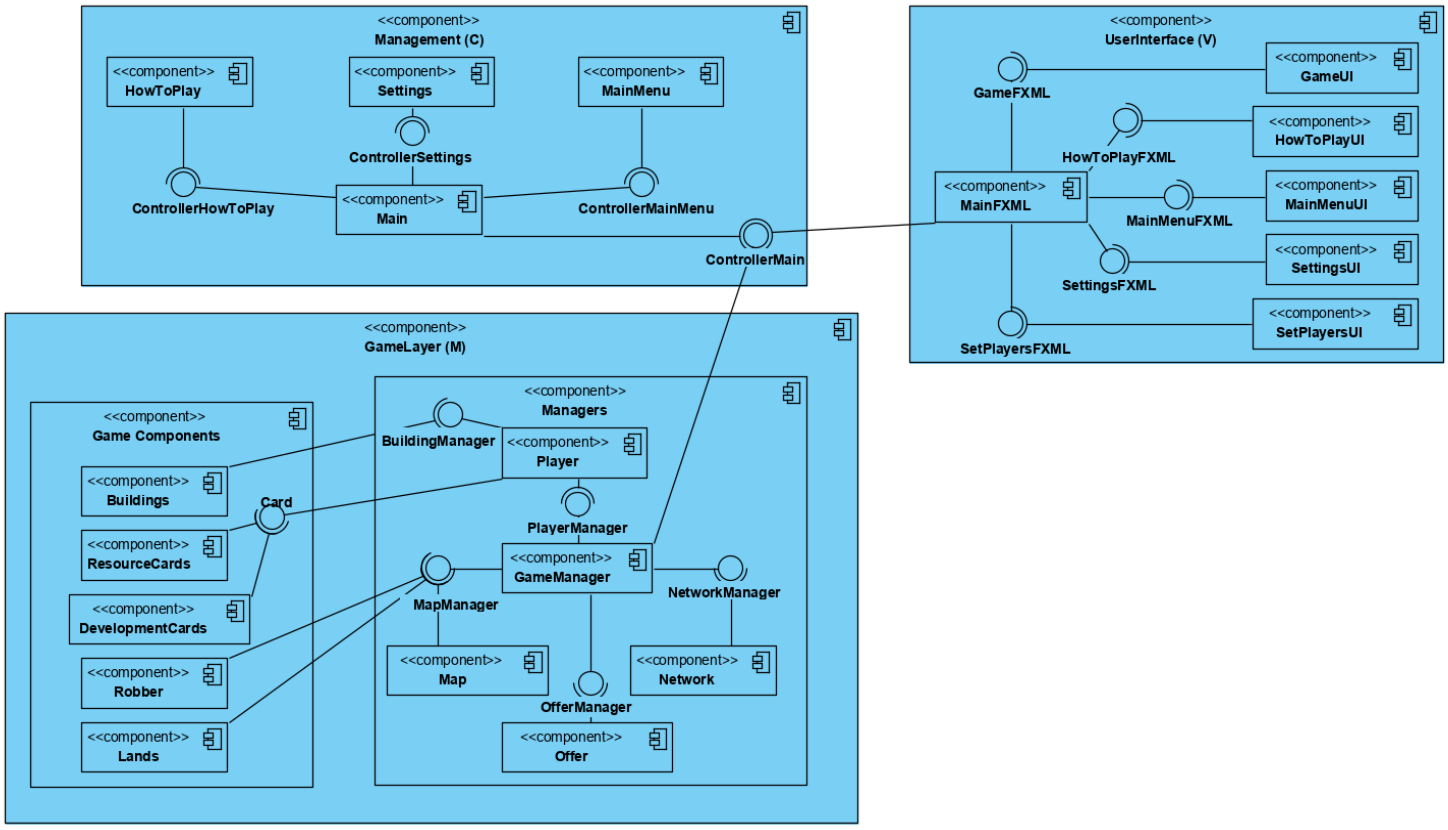

Figure 1 - Subsystem Decomposition

GameLayer Subsystem (Model): GameLayer subsystem is the core of our program and represents the “model” part of the MVC design pattern. Every active object of the game is maintained in this section through the usage of managers for each and every other part. It has been further divided into 2 main categories of “Game Components” and “Managers” in order to make it easier to comprehend.

UserInterface Subsystem (View): UserInterface subsystem is the main frame of the game. It corresponds to the “view” part of the MVC design pattern. It regulates the relation between the UI components with FXML classes and connects them to the controller.

Management Subsystem (Control): Management subsystem is the “controller” part of the MVC design pattern. It’s main goal is to manage the game though the usage of controller classes. The game will change accordingly to how these classes are maintained.

As a summary, we aimed the subsystem decomposition to have the least coupling and have the most cohesion as possible.

2.2 Hardware/Software Mapping

The “Settlers of Catan” will be a multiplayer game which can be played from different computers with LAN connection, and will be implemented in Java. Hence, all implementation will be done in software and it will be mapped to hardware through standard java libraries.

JavaFX: For user interfaces and all kind of graphical outputs will be handled throughout using JavaFX framework. Hence graphics are mapped through the monitor by JavaFX library hardware components. Also for the mouse events, JavaFX framework

provides the input that taken from low level hardware to the high level design implementation.

javax.sound.sampled: By using javax.sound.sampled package, all sound sources will be mapped through the speakers. Java Sound API is a low level design which provides the mapping the software implementation to the hardware components.

java.io: By using java.io package, data of the game will be handled in the multiplayer mode.

java.net: By using java.net package, we will handle the LAN connection input and outputs. Game will handled by peer-to-peer networking in that package.

2.3. Persistent Data Management

The “Settlers of Catan” does not playable single person hence, games does not logicly needed to save and continue in further time. The implementation of the game will be multiplayer hence game needs a server for all players. Instead of regular server networking, we will use peer-to-peer networking. The instances of the games will be stored in Random Access Memory as usual application execution.

2.4 Access Control and Security

The “Settlers of Catan” is a multiplayer game although the implementation will both have online and offline mode. In offline mode, all players should play in the same computer by sharing the same screen. Although the single computer implementation has advantages in the security side of the user as there are not any direct connections an external hardware or software.

The user can change set the game initially and all players change the settings of the game. Players can access the game in their turns which will be seen on the screen and can be able to access the playing functions of the games. There are also not any leaderboard or achievement exists in the game hence, access of the users are mostly bounded by setting the game, playing the game and command the settings.

In the online game mode, each player has the screen just for themselves and play the game like in the offline mode.

2.5 Control Flow

The “Settlers of Catan” is a four player game and single player mode will not be available in the implementation. All players enter their names from their personal computers then game starts. After initial start, each player turn is declared in the screen hence player can keep track of their turn and play the game. Game settings can also be changed by players in the game that affect only the player that change the settings.

2.6 Boundary Conditions

2.6.1 Application Setup

The “Settlers of Catan” game will have an executable file that can be playable in the IOS and Windows operating system. The system will not require any plug-ins or extension softwares. The game starts at the main menu where you can initiate a game or go into settings.

2.6.2 Terminating the Application

In order to terminate the application, a player can press the exit button in the window. The games are not saved and does not give chance to continue from the last point thus there will be an alert about that if you exit the game, all data will be lost. Hence, even if one player quits the game, game will be executed for all players.

2.6.3 Input/Output Exceptions

The game consists of only mouse events except the initial game creation. When initiating the game, each player type their names where the only input exceptions might occur. Output exceptions might also happen thus; in the implementation, most of the function that use comparison or other information about other players should include an exception.

2.6.4 Critical Error

If the game crashes, all data will be lost and game will be terminated. If any functionality of the game is not executed in the constructed way, game will also give critical error to terminate itself.

3. Subsystem Services

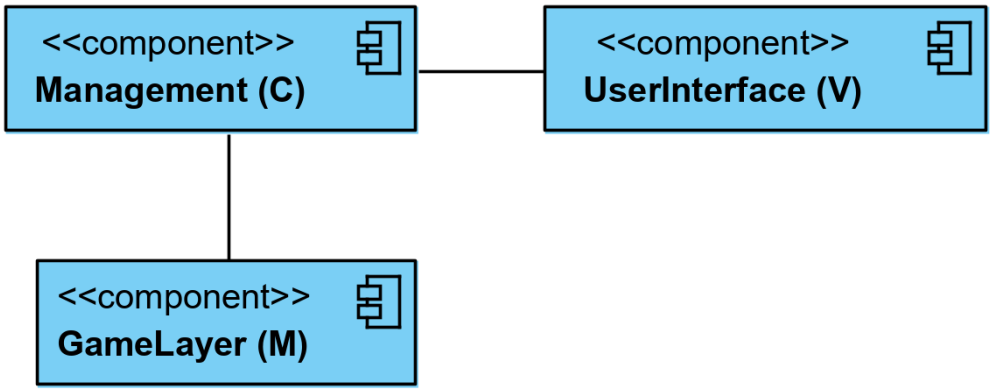

Figure 2 - Overview of the Subsystem Decomposition

(The overview of the subsystem services and the MVC design pattern can be seen in the figure above)

3.1 GameLayer Subsystem (Model)

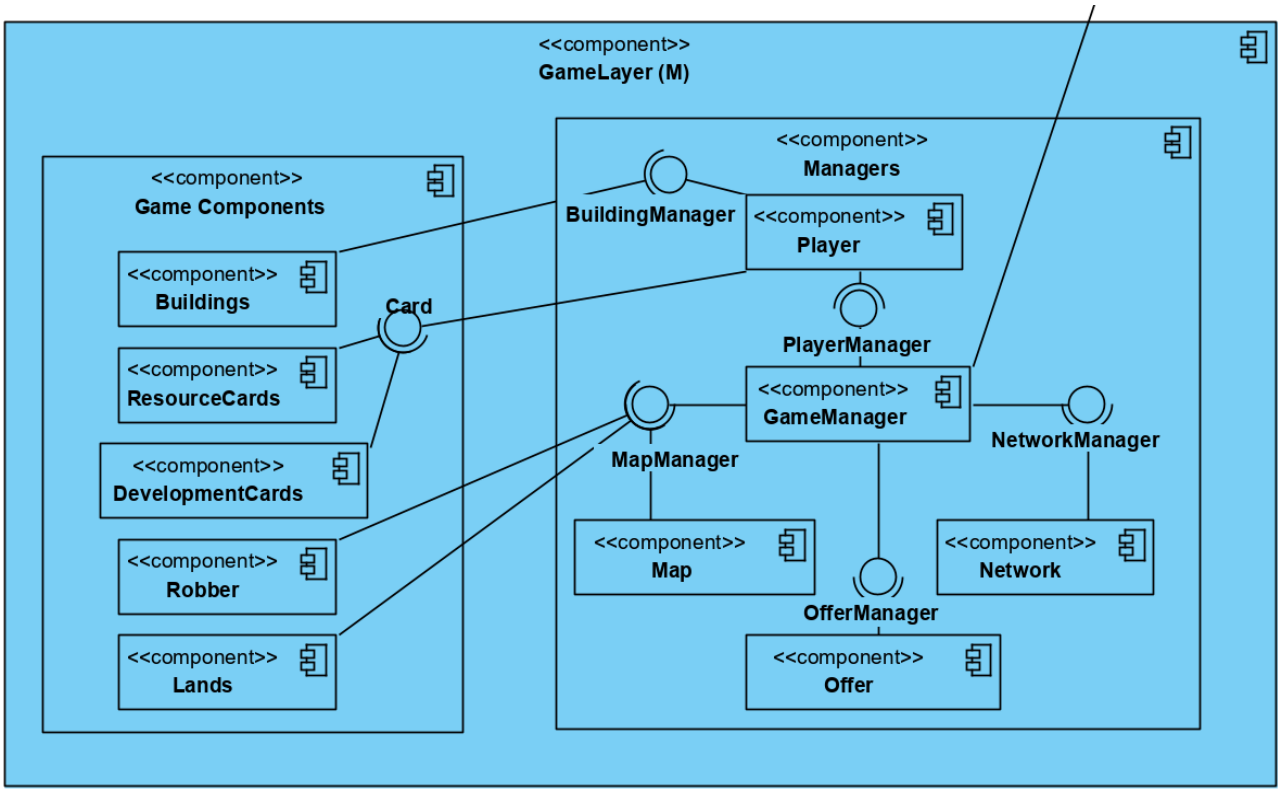

Figure 3 - GameLayer Subsystem

This subsystem is the core of the game and is responsible for all the active components of the game. It represents the “model” of the MVC pattern. The user gets to this screen via the “Play Game” button from the main menu. It connects the game to the game components through the usage of managers for each section of the game. These components consist of Buildings, ResourceCards, DevelopmentCards, Robber and Lands. All of these components are connected to the GameManager which is connected to the ControllerMain, which is updated in the Management Subsystem.

3.2 UserInterface Subsystem (View)

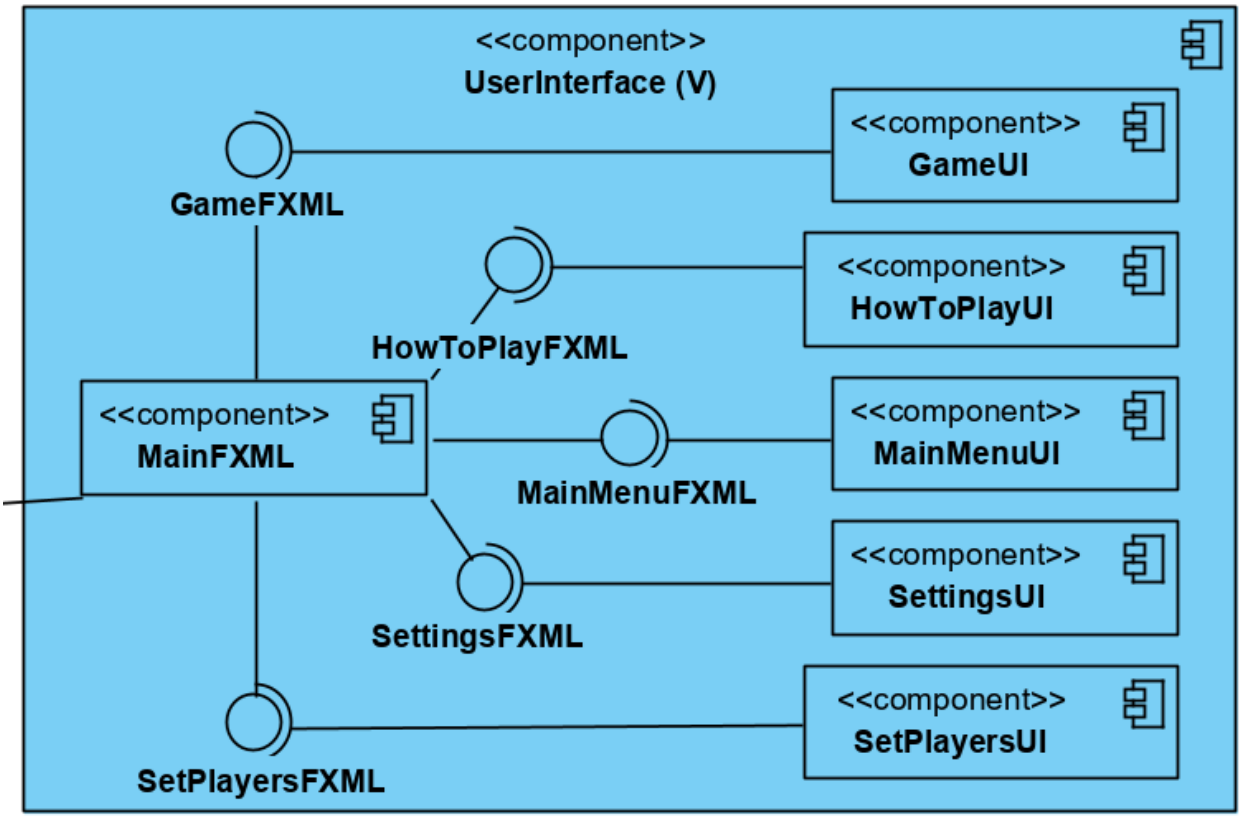

Figure 4 - UserInterface Subsystem

This subsystem is used for the display of menus to the user. This subsystem is linked with the Management Subsystem through the ControllerMain, so that when the user presses a button, the game will change accordingly. It's main goal is to provide the most user-friendly experience to the user. In order to do so, we have decided to make it as easy to use as possible with minimal complexity whilst keeping all the fundamentals functional.

In the process of designing the user interface, we have come into agreement that we should use JavaFX for both it's easy-to-use attributes as well as it's functionalities. So, these UI components are updated through FXML classes, which are JavaFX special, and we merge them in a MainFXML file.

3.3 Management Subsystem (Control)

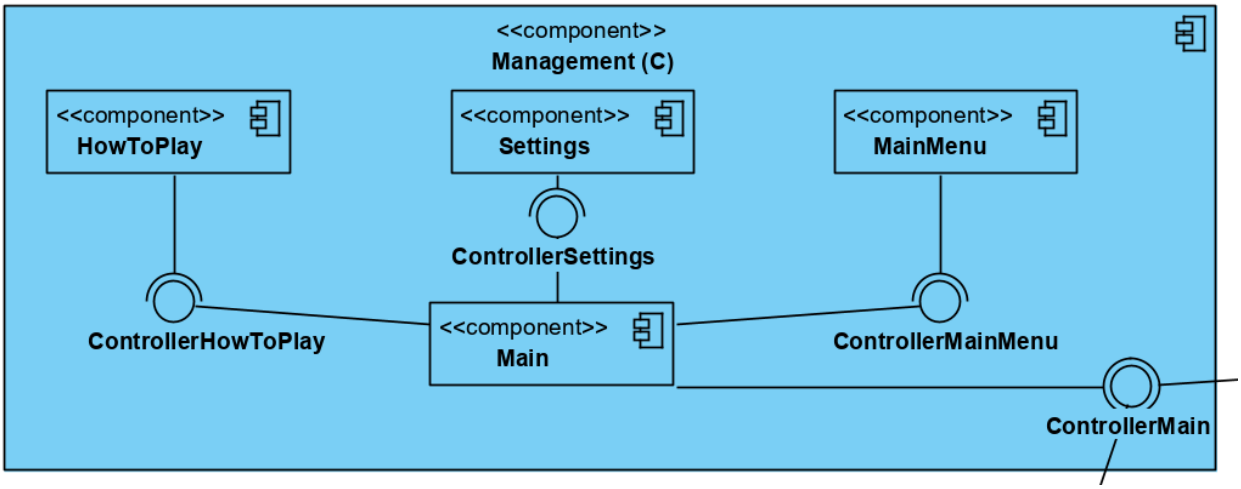

Figure 5 - Management Subsystem

This subsystem is what operates the rest as the main control point of the entire system. It updates both the GameLayer and the UserInterface subsystems through the ControllerMain. It consists of the controllers for the main menus, and a main controller for the rest of the game: GameLayer (which consists of game components and managers) and UserInterface (which consists of FXML classes).

4. Low Level Design

4.1. Design Decision and Patterns

4.1.1. MVC Design Patterns

We designed our subsystem decomposition with respect to the Model-View-Controller design pattern. This design is adequate for our case as the game will require different levels of processing. Model represents our fundamental classes which are Player, Map, Card, Offer, Trade. These are the classes which store game's data and they interact with each other as the game progresses. There are also manager classes. These classes organize the functions of the game and coordinates other classes. View section is our user-interface which interacts with players. In our design, updates appearance of the game according to the controller and model. Controller part is composed of intermediary classes. They are responsible for managing the interactions between game model and user interface.

4.1.2. Singleton Design Pattern

Our game revolves around the game grid. Resources are distributed if players has settlements around rolled number. To avoid any confusion, there should only be one game grid. All classes call the one and only instance of Map. All changes are done in the same grid. It is created when the game starts and will be used until the game is over. Same applies for our manager classes. As described above, manager classes coordinate the play mechanics which means that two instances of the same manager may cause problems. One instance may alter the game while the second one reverses the action.

4.1.3. Inheritance as an Object-Oriented Practice

In our prototype model design, we had so many similar classes and functions which may lead us to duplicate the code. Therefore, we decided to gather those similar properties together in a superior class.

Our main inheritance is the Card classes. The game of Settlers of Catan has different cards for different purposes. Our Card class is parent to classes of DevelopmentCard and ResourceCard. Only difference between the resource cards is their names. From an outsider's perspective, it may look that using instances is easier but not for our design. When building, we use these different resource cards. Therefore we need more than the instances and Card class is inherited. DevelopmentCard is also child of Card class. KnightCard, VictoryPointCard and ProgressCard are all children of DevelopmentCard class. Also RoadBuilding, YearOfPlenty and Monopoly are children of ProgressCard class.

Another inheritance that our design has is land's. There are six different land types. Forest, fields, hills, mountains, pasture and desert. We designed our model in a way that these lands are going to be extended from the class named Land. As each different land is going to give different resource, only storing the resource type inside subclasses is enough. Then we can call function of getResource(), resources of given land will be returned.

Last inheritance is the association of buildings. Road, Settlement and City classes are all built and give players some advantages. Although their profits are different than each other, if the victory points they give is stored, they can be inherited from Building class.

4.1.4 Facade Design Pattern

In order to avoid complexity of the diagrams we used facade design pattern in our model. Instead of creating every class in controllers, we instantiate only a game manager. We create every class inside this manager and process the game inside it. Therefore there will be no confusion and our model will be easy to communicate.

4.2. Final Object Design

We divide final object design in pieces to provide more readable design.

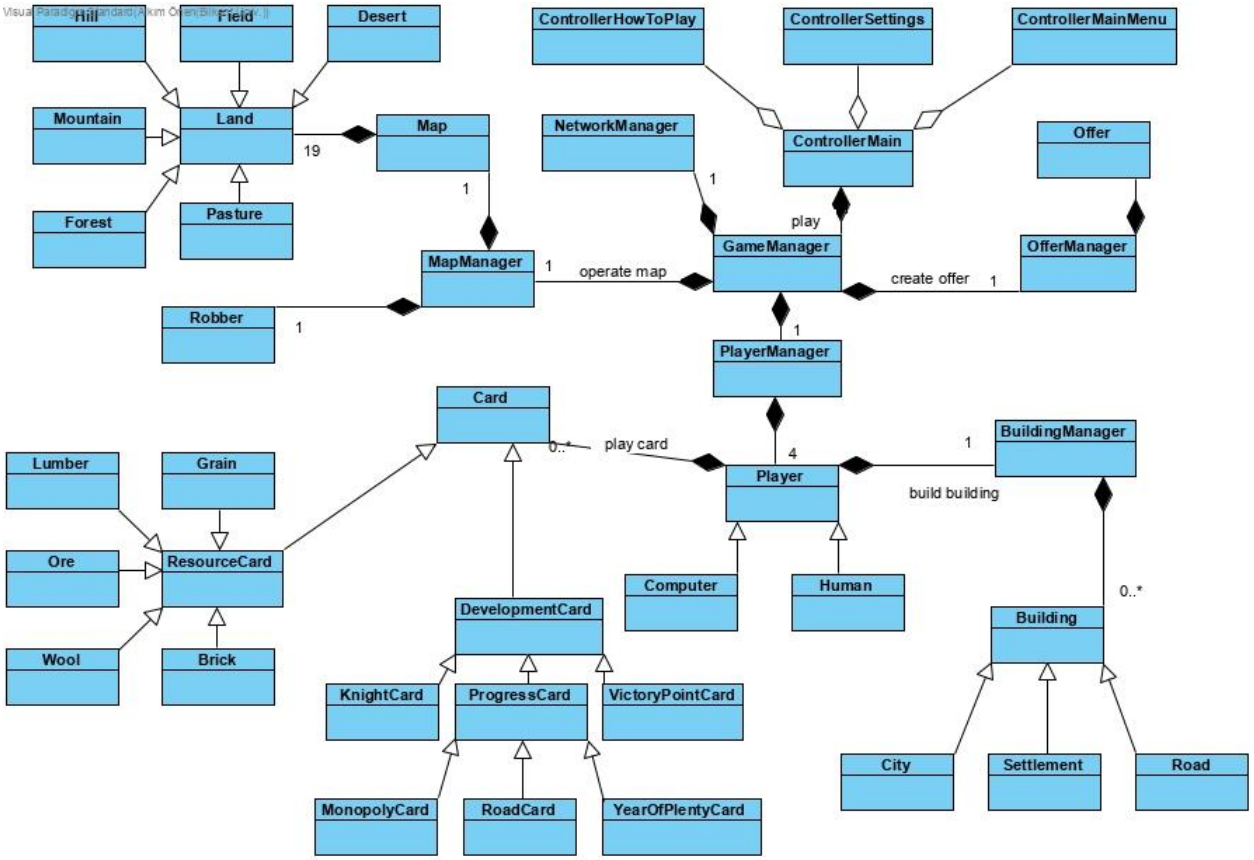

Figure 6 - Abstract Model

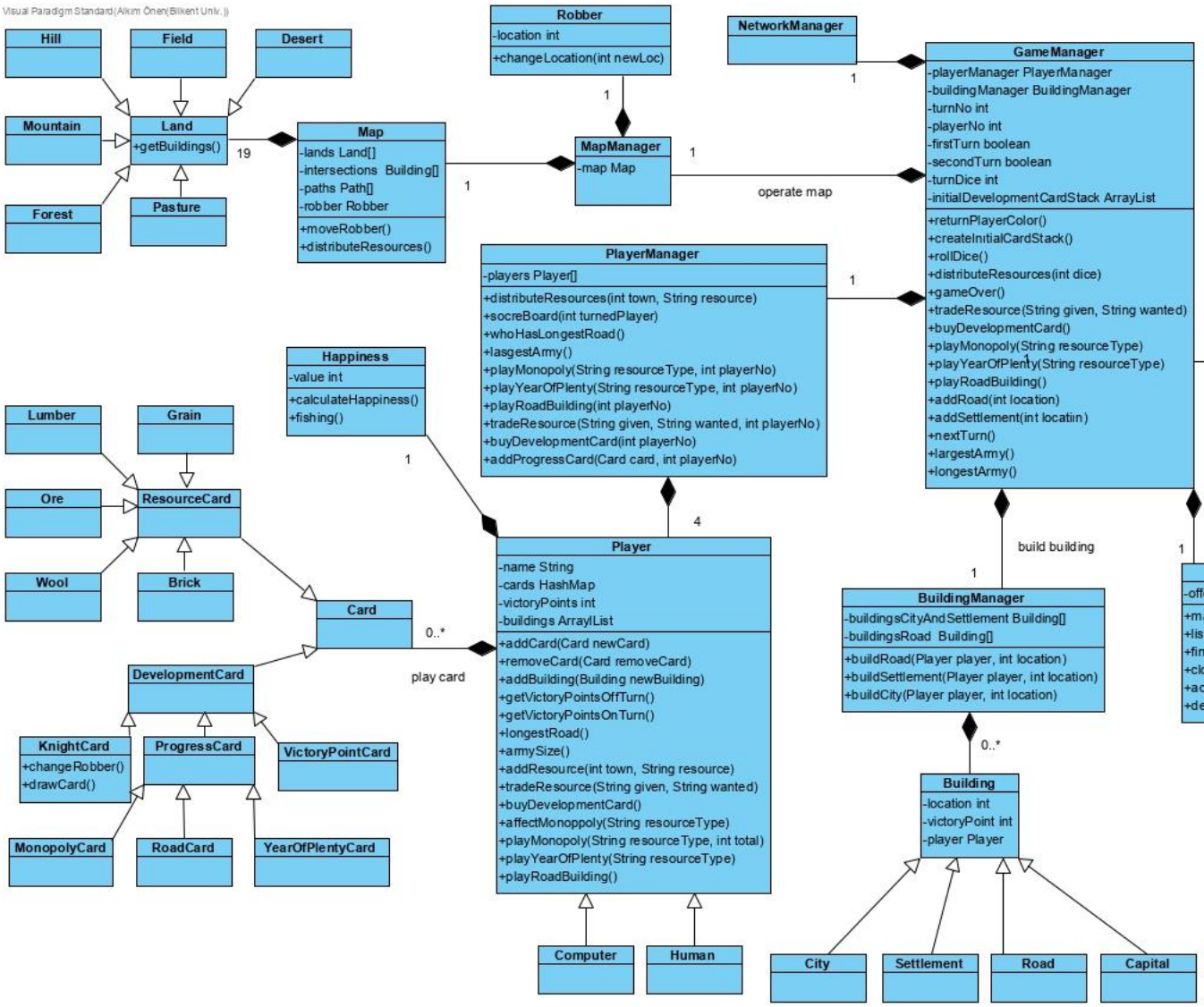
Visual Paradigm Standard(Alkım Önen(Bilkent Univ.))
Hill
Field
Desert
Mountain
Land
+getBuildings()
19
Forest
Pasture
Map
-lands Land[]
-intersections Building[]
-paths Path[]
-robber Robber
+moveRobber()
+distributeResources()
1
Robber
-location int
+changeLocation(int newLoc)
1
MapManager
-map Map
1
operate map
NetworkManager
1
GameManager
-playerManager PlayerManager
-buildingManager BuildingManager
-turnNo int
-playerNo int
-firstTurn boolean
-secondTurn boolean
-turnDice int
-initialDevelopmentCardStack ArrayList
+returnPlayerColor()
+createInitialCardStack()
+rollDice()
+distributeResources(int dice)
+gameOver()
+tradeResource(String given, String wanted)
+buyDevelopmentCard()
+playMonopoly(String resourceType)
+playYearOfPlenty(String resourceType)
+playRoadBuilding()
+addRoad(int location)
+addSettlement(int locatiin)
+nextTurn()
+largestArmy()
+longestArmy()
PlayerManager
-players Player[]
+distributeResources(int town, String resource)
+socreBoard(int turnedPlayer)
+whoHasLongestRoad()
+lasgestArmy()
+playMonopoly(String resourceType, int playerNo)
+playYearOfPlenty(String resourceType, int playerNo)
+playRoadBuilding(int playerNo)
+tradeResource(String given, String wanted, int playerNo)
+buyDevelopmentCard(int playerNo)
+addProgressCard(Card card, int playerNo)
1
4
Happiness
-value int
+calculateHappiness()
+fishing()
1
Lumber
Grain
Ore
ResourceCard
Wool
Brick
Card
0..*
play card
DevelopmentCard
KnightCard
+changeRobber()
+drawCard()
ProgressCard
VictoryPointCard
MonopolyCard
RoadCard
YearOfPlentyCard
Player
-name String
-cards HashMap
-victoryPoints int
-buildings ArrayList
+addCard(Card newCard)
+removeCard(Card removeCard)
+addBuilding(Building newBuilding)
+getVictoryPointsOffTurn()
+getVictoryPointsOnTurn()
+longestRoad()
+armySize()
+addResource(int town, String resource)
+tradeResource(String given, String wanted)
+buyDevelopmentCard()
+affectMonoppoly(String resourceType)
+playMonopoly(String resourceType, int total)
+playYearOfPlenty(String resourceType)
+playRoadBuilding()
Computer
Human
build building
1
BuildingManager
-buildingsCityAndSettlement Building[]
-buildingsRoad Building[]
+buildRoad(Player player, int location)
+buildSettlement(Player player, int location)
+buildCity(Player player, int location)
0..*
Building
-location int
-victoryPoint int
-player Player
City
Settlement
Road
Capital
1
-offers Array
+makeOffer(
+listOffers()
+findOffer(O
+closeOffer(
+acceptOffe
+declineOffe

Figure 7 - Design Model

4.3. Class Interfaces

In this section, our design is going to be described. We divided the model according to our sub-systems so that it can be more understandable. We have presented our class intercased progressively detailed and clear for any developer to understand and implement.

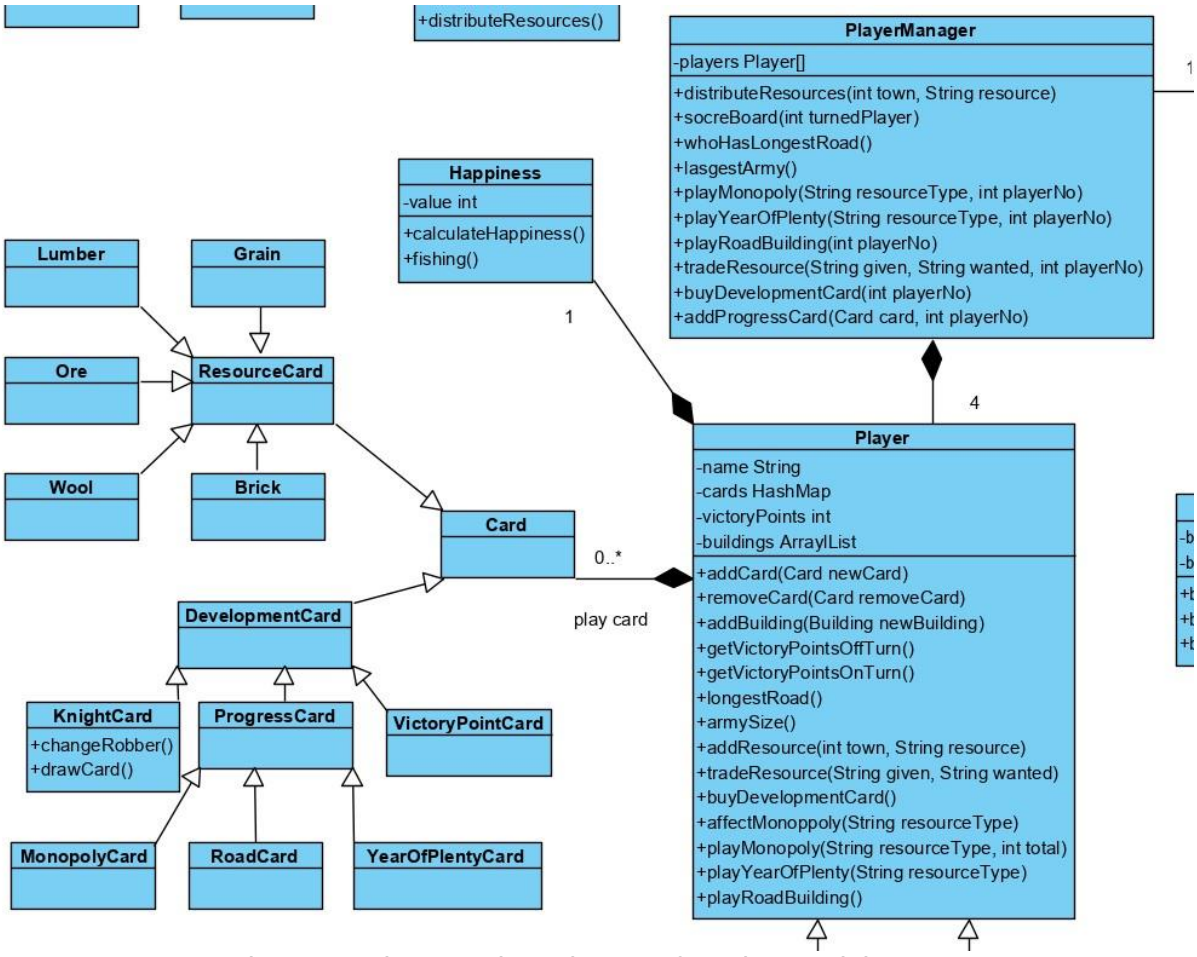

Figure 8 - Player and Card parts of Design Model

KnightCard

Methods:

- **public boolean changeRobber(int newLoc)**: Moves the robber to given location.
- **public boolean drawCard():** Makes the player to draw a card from another player that has a settlement or a city next to robber's location

Happiness

Attributes:

- **private int value:** Measurement of happiness

Methods:

- **public void calculateHappiness():** Calculates the current value of happiness of a player
- **public void fishing():** Increments happiness value with ⅓ chance

Player

Attributes:

- **private String name:** Player's name
- **private ArrayList<Building> building:** Current buildings of the player
- **private HashMap<String, int> cards:** Current cards of the player
- **private int victoryPoint:** Current victory points of the player

Methods:

- **public void addCard(Card newCard):** Adds given card to the player
- **public boolean removeCard(Card newCard):** Removes given card from the player
- **public void addBuilding(Building newBuilding):** Adds given building to the player
- **public int getVictoryPointsOffTurn():** Returns victory points of player without the Victory Point cards
- **public int getVictoryPointsOnTurn():** Returns victory points of player with the Victory Point cards
- **public int longestRoad():** Returns the longest road length of the player
- **public int armySize():** Returns the number of Knight Cards that the player has
- **public void addResource(int town, String resource):** Adds right amount of given resource card to player
- **public boolean tradeResource(String given, String wanted):** Trades given resources with the best case
- **public boolean buyDevelopmentCard():** Adds the development card at the top of initial development card stack to the player and removes according resources of purchase
- **public int affectMonopoly(String resourceType):** Removes all of the given resource cards from player and returns the number of them
- **public void playMonopoly(Strşng resourceType, int total):** Adds total number of given resource cards to player
- **public void playYearOfPlenty(String resourceType):** Adds two of given resource cards to player
- **public void playRoadBuilding()**: Removes one of the Road Building Development cards from player

PlayerManager

Attributes:

- **private Player[] players:** Contains the players of the game

Methods:

- **public void distributeResources(int town, String resource):** Distributes the given resource cards to all players that has town in given location
- **public int[] scoreBoard(int turnedPlayer):** Returns the scoreboard with the information of player turn
- **public int whoHasLongestRoad():** Returns the no of the player that has the longest road
- **public int largestArmy():** Returns the no of player that has the largest army
- **public void playMonopoly(String resourceType, int playerNo):** Plays the monopoly card for given player and resource
- **public void playYearOfPlenty(String resourceType, int playerNo):** Plays the Year of Plenty card for given player and resource
- **public void play RoadBuilding(int playerNo):** Plays the road building card for given player
- **public boolean tradeResource(String given, String wanted, int playerNo):** Trades given resource with wanted one for given player
- **public boolean buyDevelopmentCard(int playerNo):** Draws the top development card from initial stack for given player
- **public void addProgressCard(Card card, int playerNo):** Adds the given card to given player

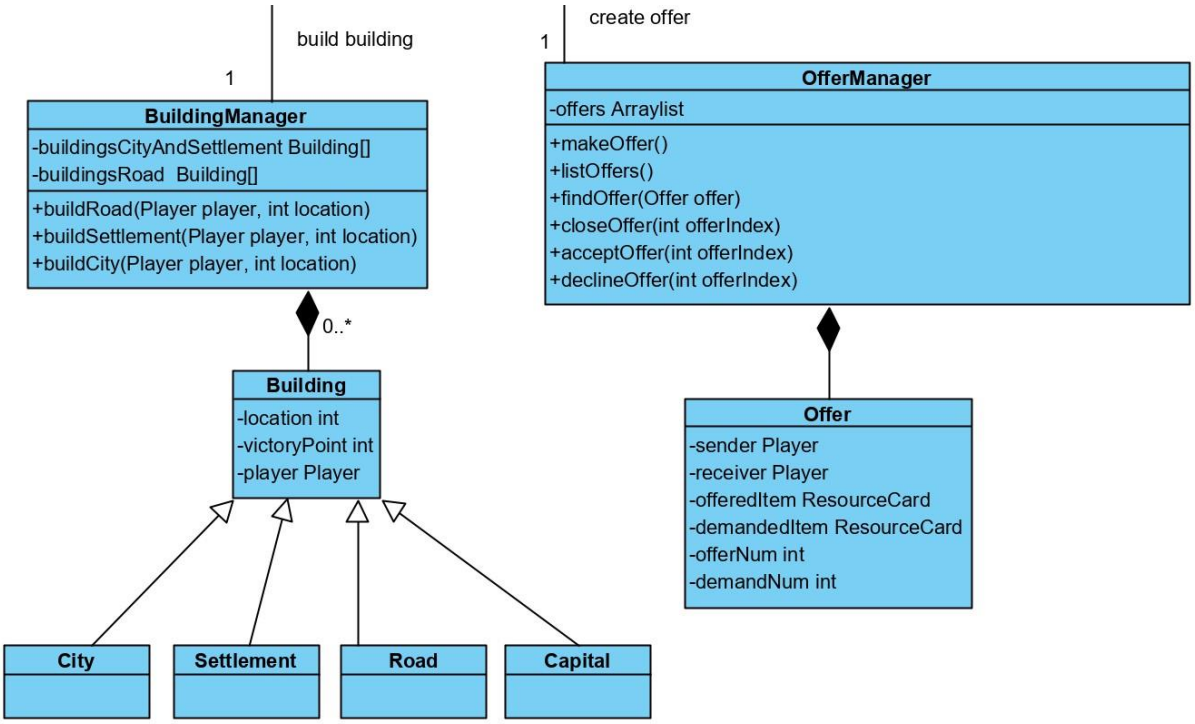

Figure 9 - Build and Offer parts of Design Model

Building

Attributes:

- **private int location:** Location of the building in the grid
- **private int victorPoint:** Number of Victory Points that the building values
- **private Player player:** Owner of the building

BuildManager

Attributes:

- **private Buildings[] buildingsCityAndSettlement:** Contains all the buildings on the intersections
- **private Building[] buildingsRoad:** Contains all the buildings on paths

Methods:

- **private boolean buildRoad(Player player, int location):** Builds a road for that player and update the buildings location and players building attributes
- **private boolean buildCity(Player player, int location):** Builds a city for that player and update the buildings location, players building attributes and player manager to update the players victory points
- **private boolean buildSettlement(Player player, int location):** Builds a settlement for that player and update the buildings location, players building attributes and player manager to update the players victory points

Offer

Attributes:

- **private Player sender:** Player that sent the offer
- **private Player receiver:** Player that the offer is sent
- **private ResourceCard offeredItem:** Offered item's resource card.

- **private ResourceCard demandedItem:** Demanded item's resource card
- **private int offerNum:** Number of resources offered.
- **private int demandNum:** Number of resources demanded.

OfferManager

Attributes:

- **private ArrayList<Offer> offers:** All the user offers inside an array of offer objects

Methods:

- **private void makeOffer():** Create new offer and stack it in offers array where the correspondent of the will be able to view it
- **private void listOffers():** List the relevant offers for that player from all offer array
- **private boolean findOffer(Offer offer):** Returns the relevant offer for that player from all offer array
- **private void closeOffer(int offerIndex):** Close all the offers for that user in that turn
- **private boolean acceptOffer(int offerIndex):** Accept that offer instance and inform player managers player array
- **private boolean declineOffer(int offerIndex):** Cecline that offer and delete that offer instance from offer array

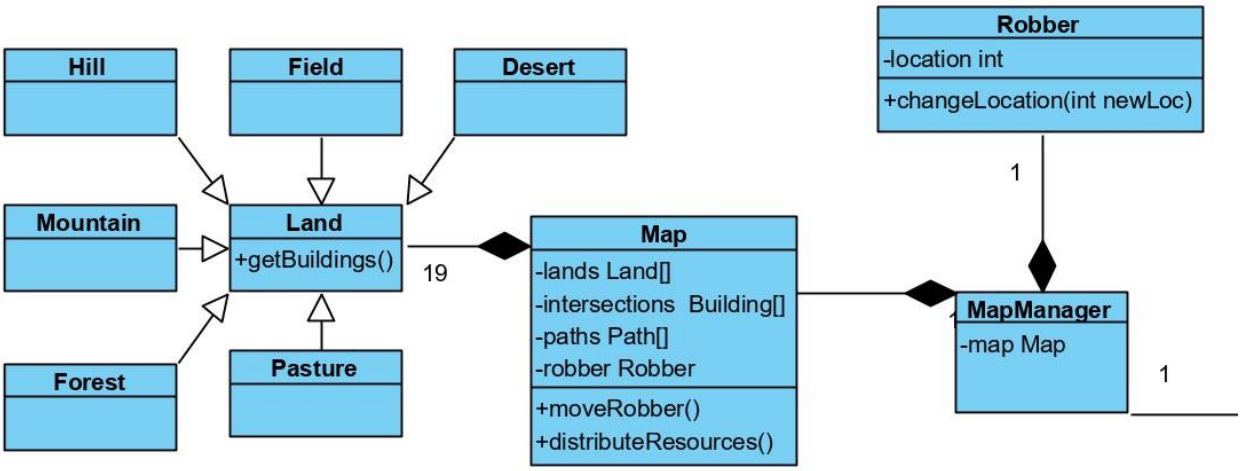

Figure 10 - Grid part of Design Model

Land

Methods:

- **Public Building[] getBuildings():** Returns all of the buildings on the grid

Map

Attributes:

- **private Land[] lands:** Lands on the grid.
- **private Building[] paths:** Buildings built on paths.
- **private Building[] intersection:** Buildings built on intersections.
- **private Robber robber:** Robber instance

Methods:

- **private void distributeResources(int dice):** Distributes the resources according to the rolled dice
- **private boolean moveRobber():** Moves the robber in the game grid when needed

Robber

Attributes:

- **private int location:** The location of robber on the grid

Methods:

- **private boolean changeLocation(int newLoc):** Changes the location of robber to given location on grid

MapManager

Attributes:

- **private Map map:** Map instance for the game grid

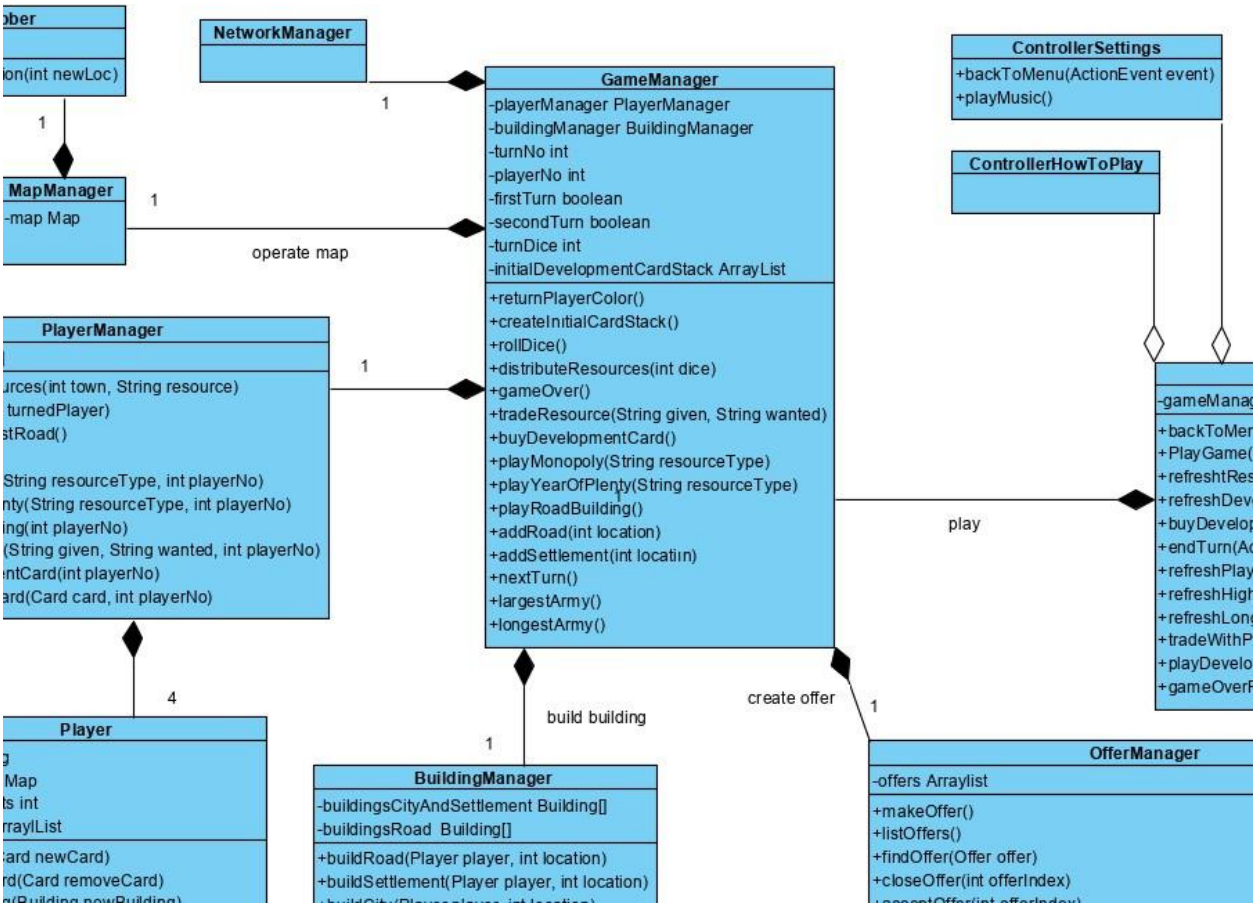

Figure 11 - Game Manager part of Design Model

GameManager

Attributes:

- **private PlayerManager playerManager:** Instance of Player Manager
- Private BuildingManager buildingManager: Instance of Building Manager

- **private int turnNo:** Number of turns that passed since the start of the game
- **private int playerNo:** Number that represents which player's turn it is
- **private boolean firstTurn:** Boolean that represents if it is the first turn or not.
- **private boolean secondTurn:** Boolean that represents if it is the second turn or not.
- **private int turnDice:** Rolled dice
- **private Card[] initialDevelopmentCardStack:** The randomized stack of Development Cards.

Methods:

- **public String returnPlayerColor():** Returns the color of current player
- **private ArrayList<Card> createInitialCardStack():** Shuffles a brand new stack of development cards
- **public int rollDice():** Returns a number between 1 and 12
- **public boolean distributeResources(int dice):** Distributes the resources to the players that have towns next to rolled dice
- **public Player gameOver():** Returns the winner of the game
- **public boolean tradeResource(String given, String wanted):** Trades given resource with wanted resource with calculating the best case for current player
- **public boolean buyDevelopmentCard():** Buys a development card for current player if he/she has enough resources
- **public void playMonopoly(String resourceType):** Plays the Monopoly development card for current player if he/she has at least one
- **public void playYearOfPlenty(String resourceType):** Plays the Year of Plenty development card for current player if he/she has at least one
- **public void playRoadBuilding():** Plays the Road Building development card for current player if he/she has at least one
- **public boolean addRoad(int location):** Builds a road in given location if path is empty
- **public boolean addSettlement(int location):** Builds a town in given location if intersection is empty, upgrades building if it is not empty and current player has a town on it
- **public String nextTurn():** Moves turn to the next player
- **public int largestArmy():** Returns the no of player that has the largest army
- **public int longestRoad():** Returns the no of player that has the longest road

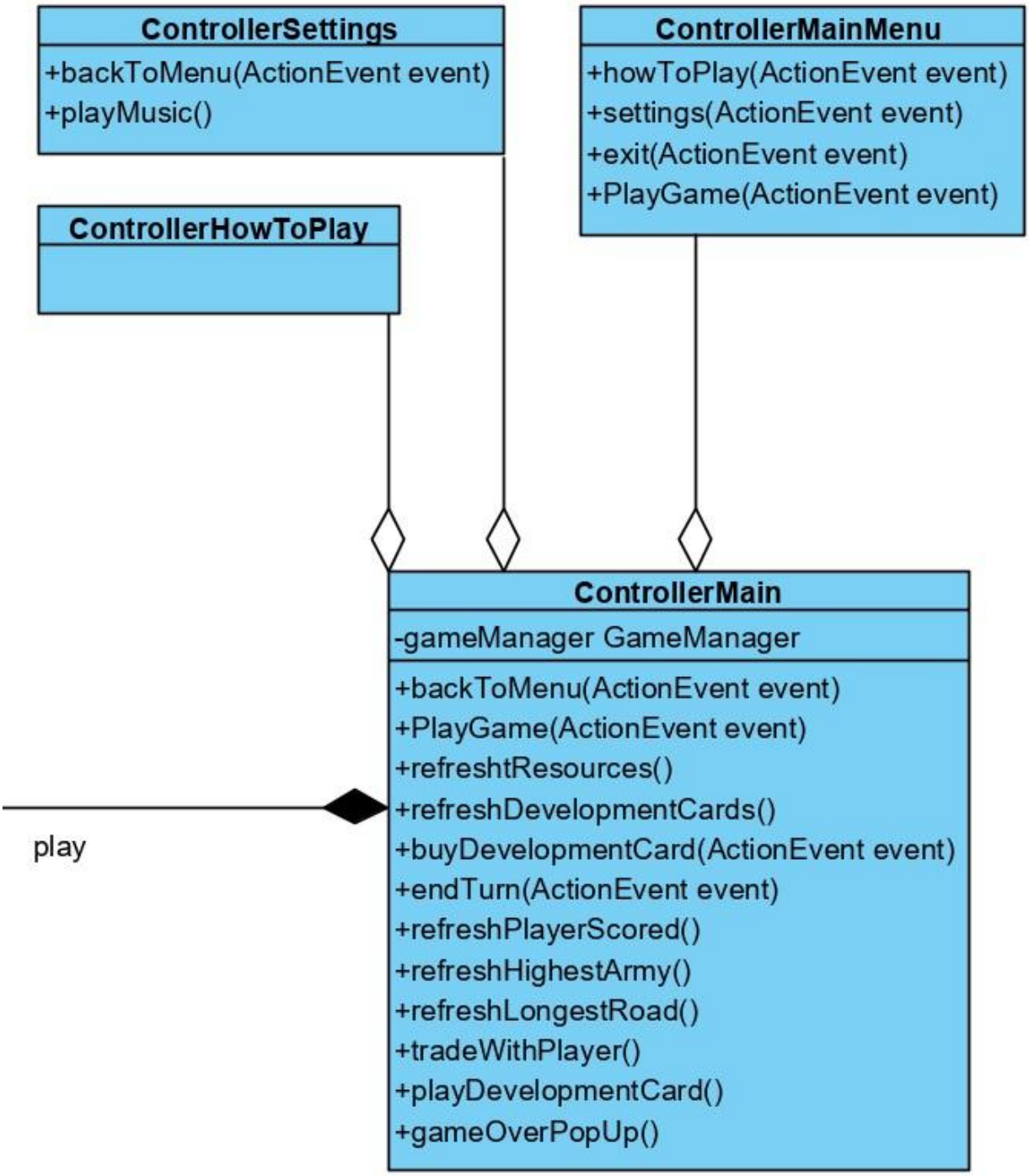

Figure 12 - Controller Section of Design Model

ContollerMain

Attributes:

- **private GameManager gameManager:** Game manager instance for the game of Catan

Methods:

- **public void backToMenu(ActionEvent event):** Changes current view to the menu frame
- **public void playGame(ActionEvent event):** Initializes game manager with given information

- **public void refreshResources():** Refreshes current resource card frame
- **public void refreshDevelopmentCards():** Refreshes current development cards fram
- **public void buyDevelopmentCard(ActionEvent event):** Buys a development card for current player
- **public void endTurn(ActionEvent event):** Ends the turn of current player
- **public void refreshPlayerScores():** Refreshes the victory points on scoreboard
- **public void refreshHighestArmy():** Refreshes the largest army part of scoreboard
- **public void tradeWithPlayer():** Trades selected resources for current player
- **public void playDevelopmentCard():** Plays selected development card with selected information
- **public void gameOverPopUp():** Informs all players that the game is over and announces the winner

ControllerMainMenu

Methods:

- **public void howToPlay(ActionEvent event):** Changes the view to how to play frame
- **public void settings(ActionEvent event):** Changes the view to settings frame
- **public void exit(ActionEvent event):** Exits the game
- **public void playGame(ActionEvent event):** Changes the view to set player names frame

ContollerSettings

Methods:

- **public void backToMenu(ActionEvent event):** Changes view to main menu frame
- **public void playMusic():** Enables or disables the music according to user's selection

4.4. Packages

The packages that implementation will consist can divided into two as internal and external packages. Internal package will consist of the code files that will be implemented. External package will map the software of the implementation to the hardware (2.2).

4.4.1 Internal Subsystem Packages

The internal packages will be divided into systems (2.1). This way it makes implementation easier and provides better monitoring for the system. The following subsystems will be *Management*, *UserInterface* and *GameLayer.* The partition of the system also provides better participation in the implementation stage.

4.4.1.1 Management

Management subsystem will consist of all the game managers such as PlayerManager, OfferManager, MapManager, BuildManager, TradeManager, OfferManager. These managers will contribute the main flow of the gameplay.

4.4.1.2 UserInterface

UserInterface subsystem will consist of all the interface and where the user intends to create a new game.

4.4.1.3 GameLayer

This system will use the Management subsystem to evaluate the main flow of the game and control the game dynamics.

4.4.2 External Packages

There are plenty of external packages for our implementation to map it from software to hardware as already mentioned (2.2). Other Java external packages might be necessary for the implementation like *java.util* but all of them are in the Java Standard Library. For applying peer to peer network, we will also use javas library; *java.io* and *java.net*. As we may proceed on the network features, we may use some other external packages, which we will mention them in the final design report.

5. Improvement Summary

In the first iteration game could be played on one computer with four players. For the second iteration we will do the game as multiplayer game that every player can play it with their own computers in the same network.

The multiplayer feature of the game needs many changes in the UI in main game grid. As an example, every player will see their own resource cards for every turn, beforehand, the resource card part of the screen was changing for every player. Moreover, we will improve the graphics of the game. The roads and buildings were the colorful buttons, we will improve the appearance of roads and buildings which makes them more understandable. In addition, game grid was inanimate, we will improve the view of the game grid which will be more elegant. Also, we will change the themes of the game which was looking bad. Game grid will also not consist of a full image where all hexagons will be created randomly and they will be independent in the new game grid.

For the game features, we will add happiness for every player and fishing feature. Players who have any building near the docks have a fixed chance to catch a fish for his own turn. The number of resource cards and fish that the players have will determine the happiness of the players. The capital will also increase the happiness and for each player, and a capital is a must for players. For the construction; happiness will decrease by the disjoint of the settlements, which will conclude to loss of cards. The player who has the most happiness will gain extra victory points. We will introduce the happiness and capital mechanisms to increase the importance of the docks and reduce the power of disjoint settlement approach in the game where in our analysis, it is the most overpowered strategy in the game. Also we want to add some random number generator effect to switch the game a little bit from the actual real life board game. All of these rule changes and improvements will be declared in the how to play menu as well.

6. Glossary & references