Bilkent University

Department of Computer Engineering

CS 319 - Object-Oriented Software Engineering

Project Design Report

Iteration 1

Settlers of Catan

Group 1D

Berke Oğuz

Alkım Önen

Kaan Tapucu

İbrahim Eren Tilla

Hasan Yıldırım

Table of Contents

1.Introduction

1.1 Purpose of system

Catan is a multiplayer strategy board game. Its plan is actualized in such ways that the clients could get the conceivable greatest fulfillment from the game. Catan game is wanted to be a convenient, easy to understand and difficult game which means to engage the clients by including them into the board game. The player's objective is to surpass all the opponents by accessing 10 points. It means to be sufficient quick, easy to use and with high frame per second.

1.1.1 Trade-Offs

1.1.1.1. Development Time vs Performance

A Large portion of individuals prefer that Java has the best option because of large community that can help each other and flexibility of java. Furthermore, unlike the C++, we did not messed with memory leaks that will reduce the development time.

1.1.1.2. Functionality vs Usability

Catan is a game that has basic controls. Since we are making a game with not complicated controls that mostly will be made by mouse clicking, we preferred ease of use over usefulness. Our game is extremely easy to comprehend, and furthermore has a how to play menu that provides users better understanding.

1.1.1.3. Understandability vs Functionality

As it was referenced previously, our framework is relied upon to be anything but difficult to learn and justifiable game. In regard to this, we needed to diminish the multifaceted nature and

usefulness of the game by wiping out confounding and convoluted game materials. As a result, the players would not try to understand, yet would appreciate the straightforward reasonable game.

1.1.1.4. Memory vs Maintainability

During the design and the analysis of the game, we tried to utilize deliberation. Our principal objective in this plan believing was to keep up the game questions 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 client experience. By profiting these regular properties of subclasses, players will be controlled with effectively.

1.1.2 Criteria

1.1.1.5. 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 utilize with focusing on the game itself, instead of understanding the game capacities. The framework will get the console contributions from the clients, 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 distinguish and see how the turns is finished, when the cities, roads or towns are built, trades, gathering resources and so on.

1.1.1.6. 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.

1.1.1.7. 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.

1.1.1.8. Modifiability

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

1.1.1.9. Reusability

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

1.1.1.10. Portability

In light of the fact that portability grows the scope of the game's ease of use for clients which is a significant point for a product when all is said in done. As it were, it causes the game to be played in different devices. Along these lines, we chose to actualize in Java, since its JavaFX lets the stage to be free and framework to be compact.

1.1.1.11. Performance Criteria

It is significant for the game to have the quick responses for the players' solicitations. Catan game is planned so its responses were practically prompt, 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 for 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. 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:

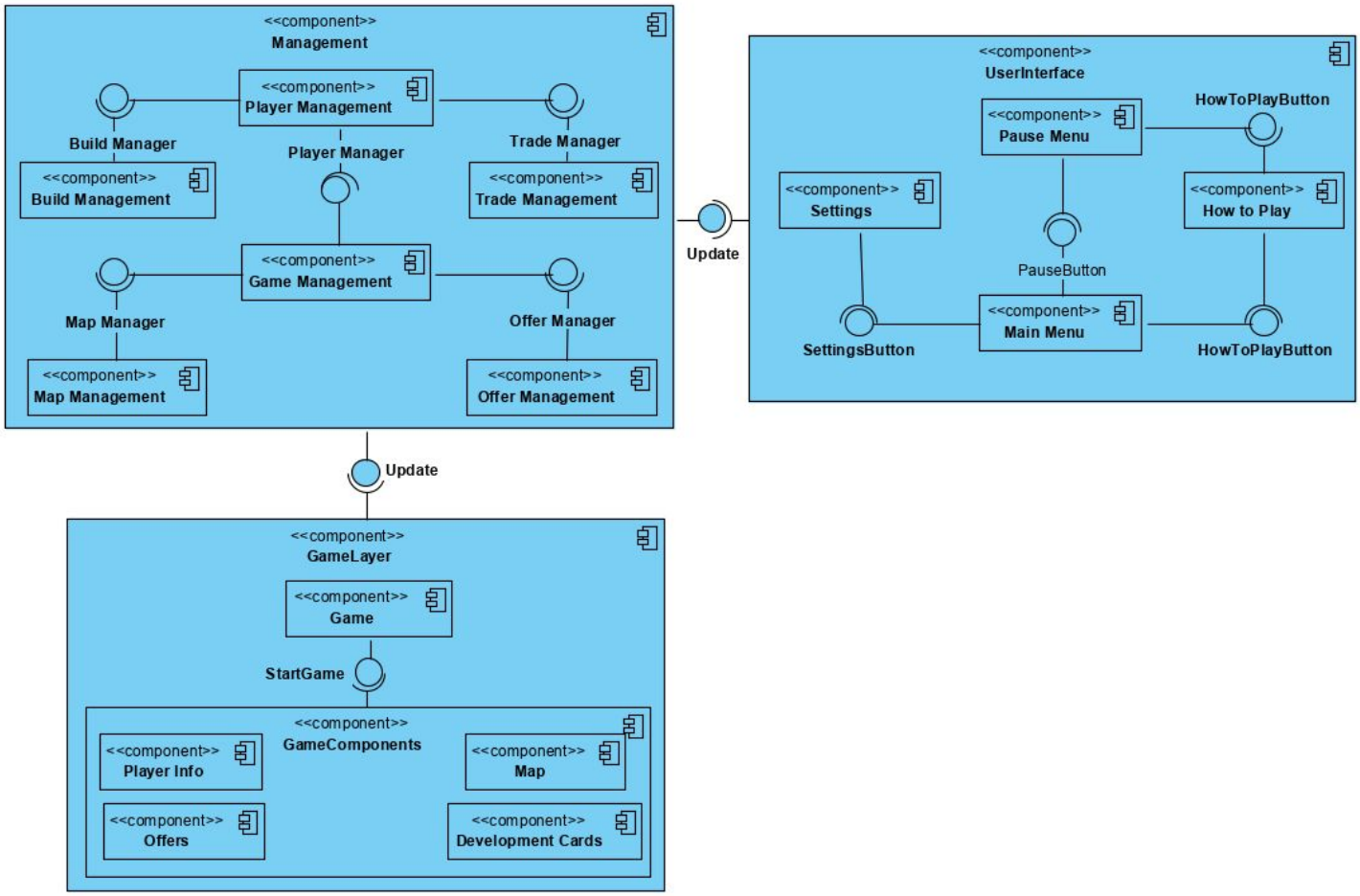

Figure 1 - Subsystem Decomposition

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.

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 menus, buttons and listeners.

Management Subsystem (Control): Management subsystem is the “controller” part of the MVC design pattern. It’s main goal is to manage how the users/players interact with the software. The game will change accordingly to what input is given.

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 neither be an online game nor a game where you can continue after quitting, 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.

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. Also, the implementation is not online multiplayer hence it does not need any database system. 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 not be an online game. Thus, 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.

2.5 Control Flow

The “Settlers of Catan” is a three to four player game and single player mode will not be available in the implementation. The game is created initially by one user and all players enter their names 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 all players in the game.

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 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

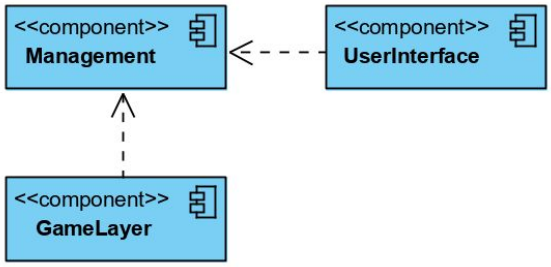

Figure 2 - Overview of the Subsystem Decomposition

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

3.1 GameLayer Subsystem (Model)

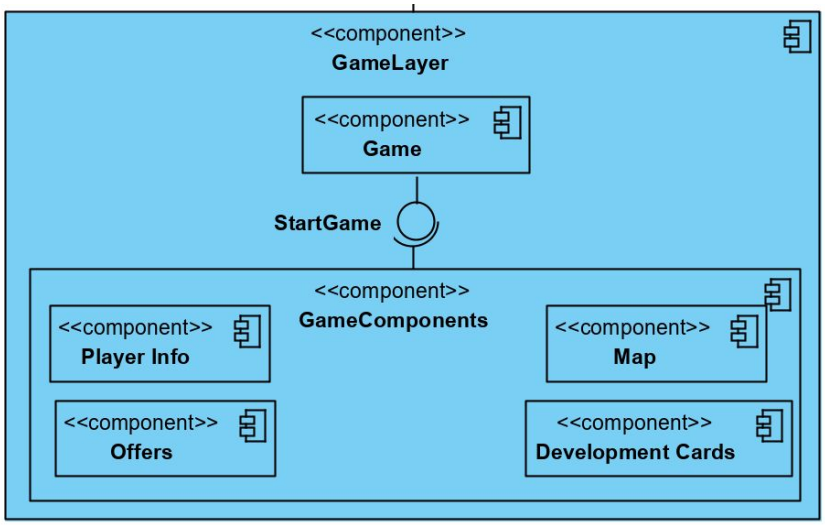

Figure 3 - GameLayer Subsystem

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. These components consist of Player Info, Map, Offers and Development Cards. The GameLayer is updated through Management subsystem.

- **Player Info:** Shows the name of the player as well as his game summary (points, trades etc.)
- **Offers:** Shows the offers list and the available players and resources a player can make an offer.
- **Development Cards:** Shows the development cards list. Hints the user the development cards that are available to play.
- **Map:** Shows the map where all the crucial things come into play. The buildings, the robber, trading harbors and the lands are shown in this part.

3.2 UserInterface Subsystem (View)

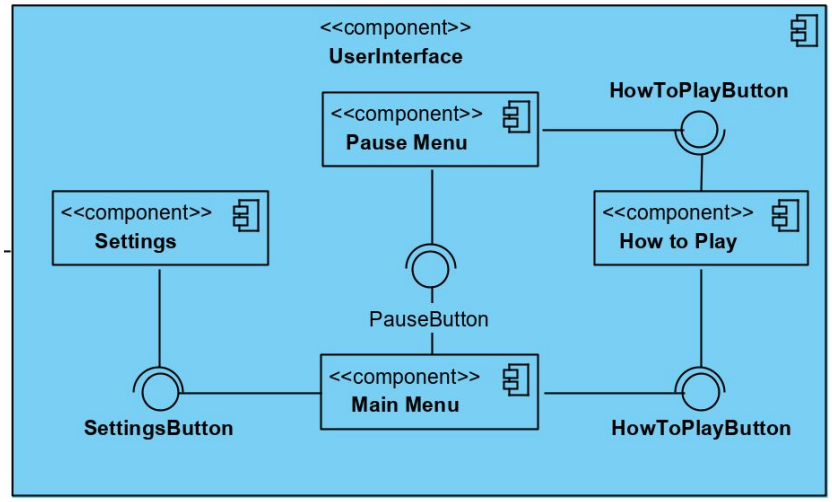

Figure 4 - UserInterface Subsystem

This subsystem is used for the display of menus to the user. This subsystem is linked with the managers 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. It consists of Main Menu, Pause Menu, Settings and How to Play.

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. The UserInterface is updated through Management subsystem.

3.3 Management Subsystem (Control)

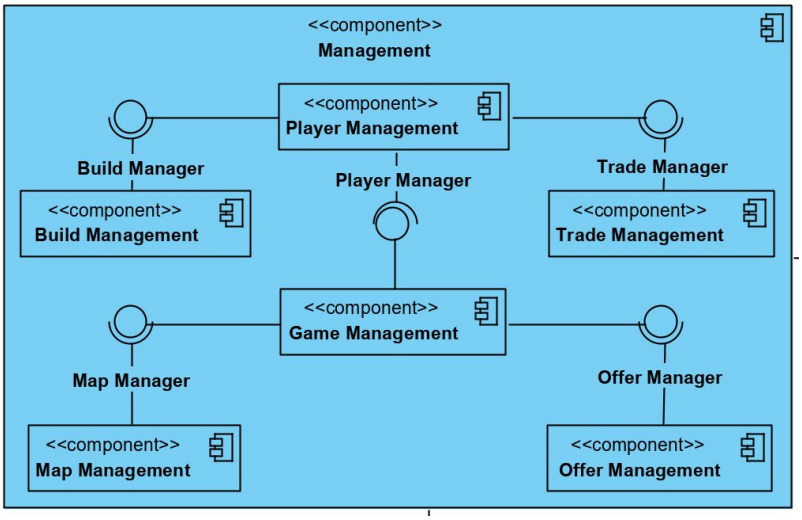

Figure 5 - Management Subsystem

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. It consists of the managers for all different aspects of the game: Game Management (which manages the whole game through Game Manager), Map Management (which consists of a Map Manager), Offer Management (which consists of an Offer Manager) and Player Management (which consists of a Player Manager).

The Player Management is where all the directly player-related actions are taken care of. It has two of its sub-components of its own: Trade Management (which consists of a Trade Manager) and a Build Management (which consists of a Build Manager).

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. Controller part is mainly our manager classes. These classes organize the functions of the game and coordinates other classes. View section is our user-interface. In our design, updates appearance of the game according to the controller and model.

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 Façade Design Pattern:

In order to avoid complexity of the diagrams we used façade design pattern. Progress card class is one of the examples that we used façade design pattern. We hold the three different types of progress cards as the subclasses of progress card class instead of combining three different cards directly to development card class. Moreover, resource card class has the same feature because it holds the subclasses of resource cards.

4.2. Final Object Design

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

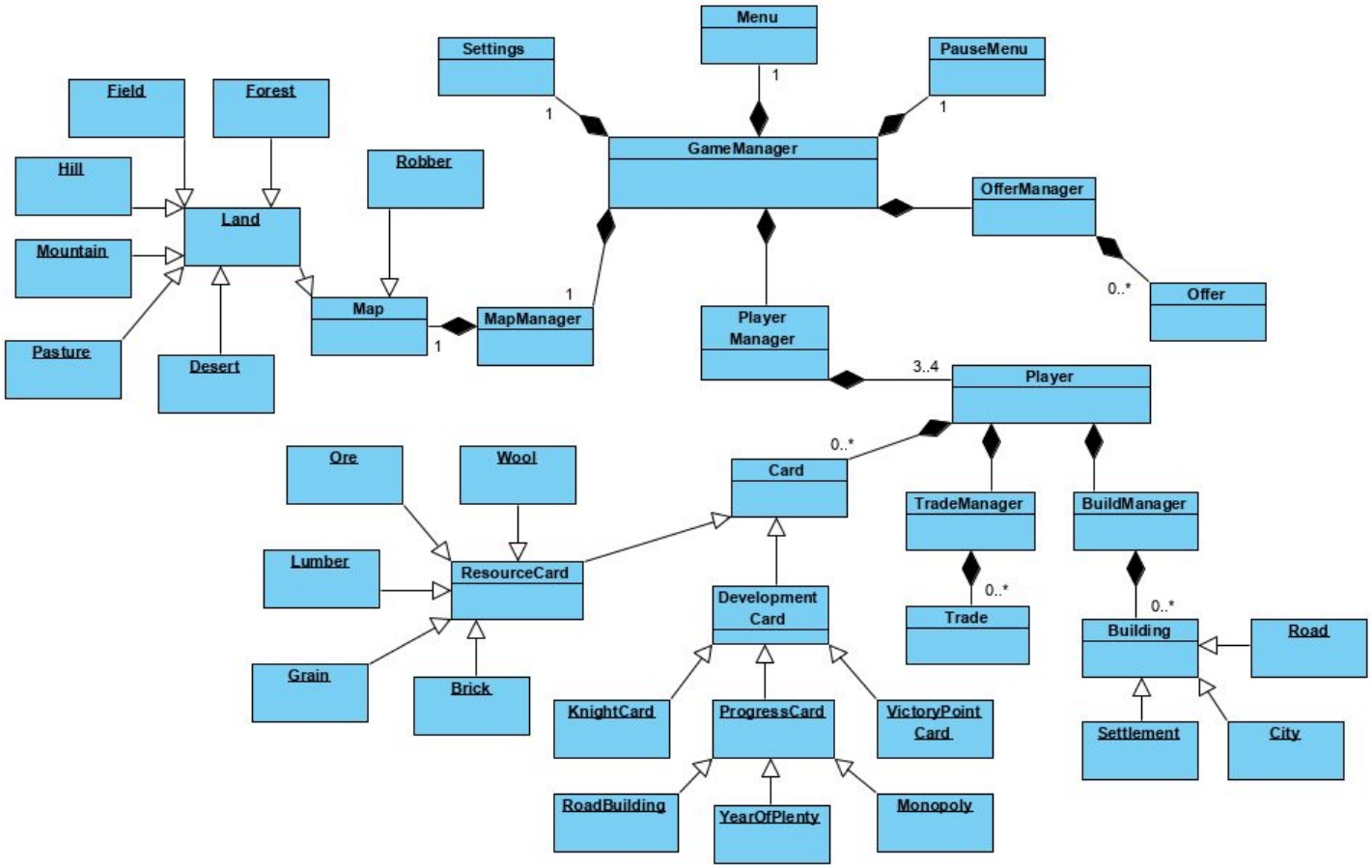

Figure 6 - Abstract Model

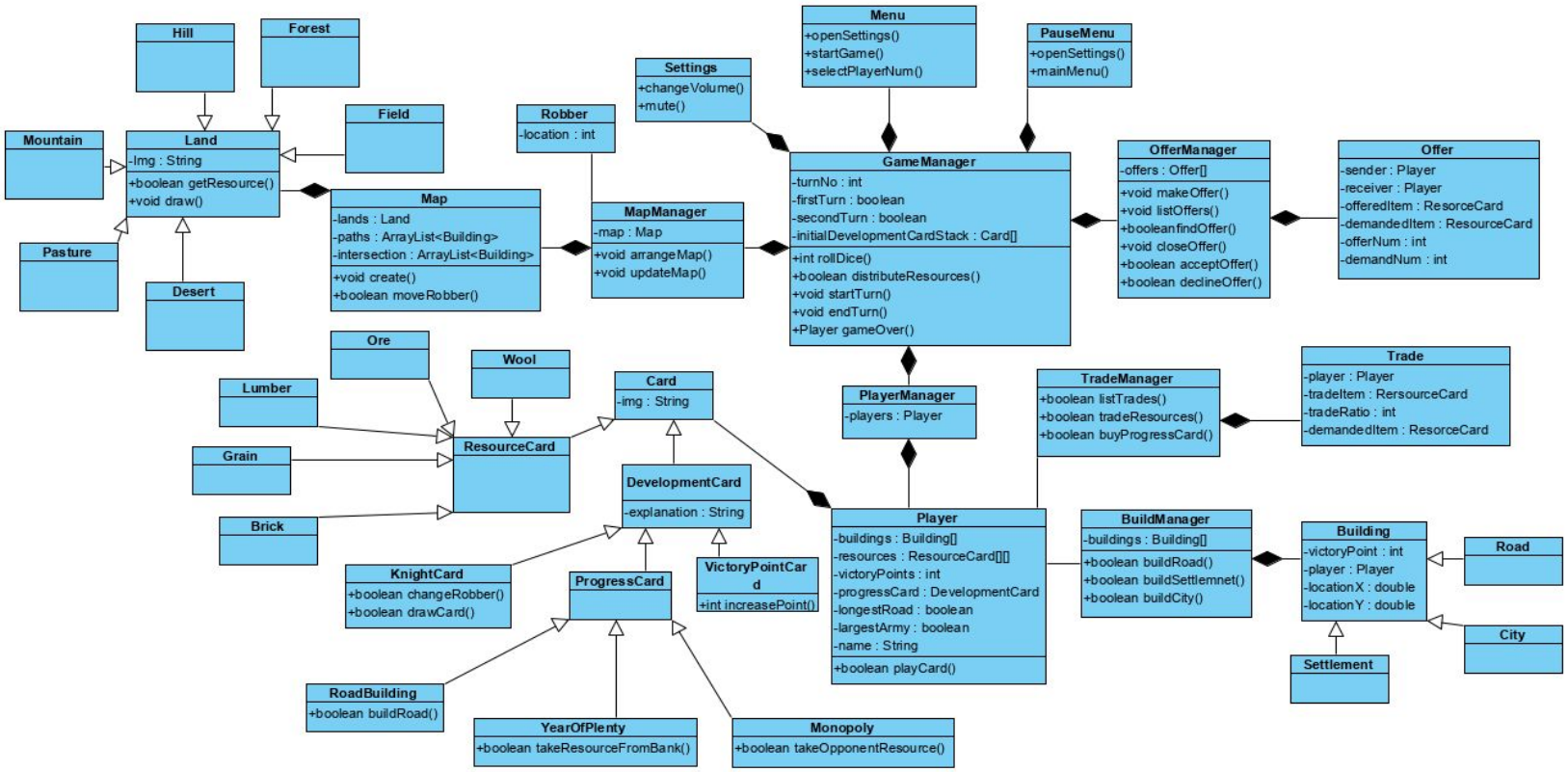

Figure 7 - Design Model

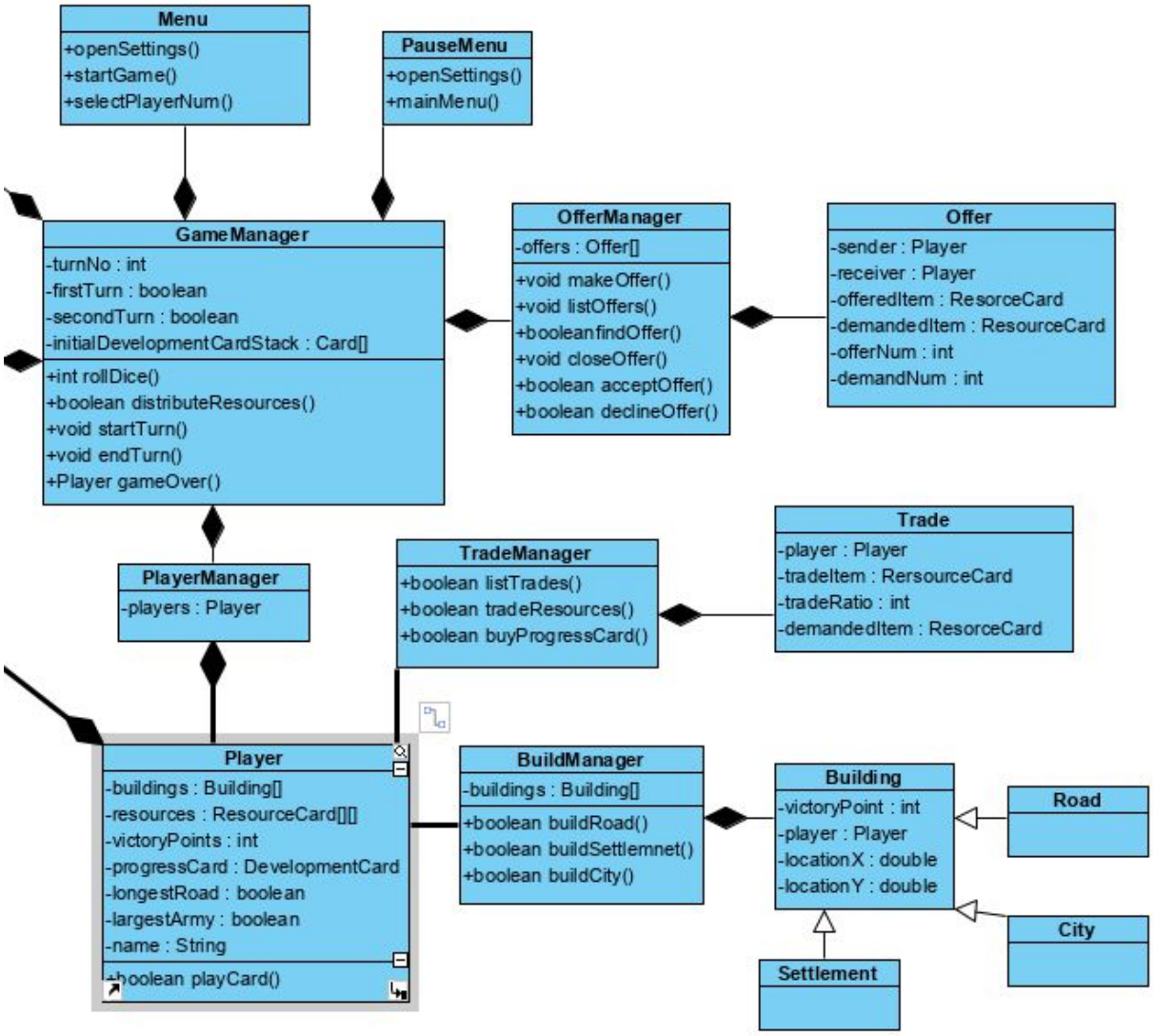

Figure 8 - First part of Design Model

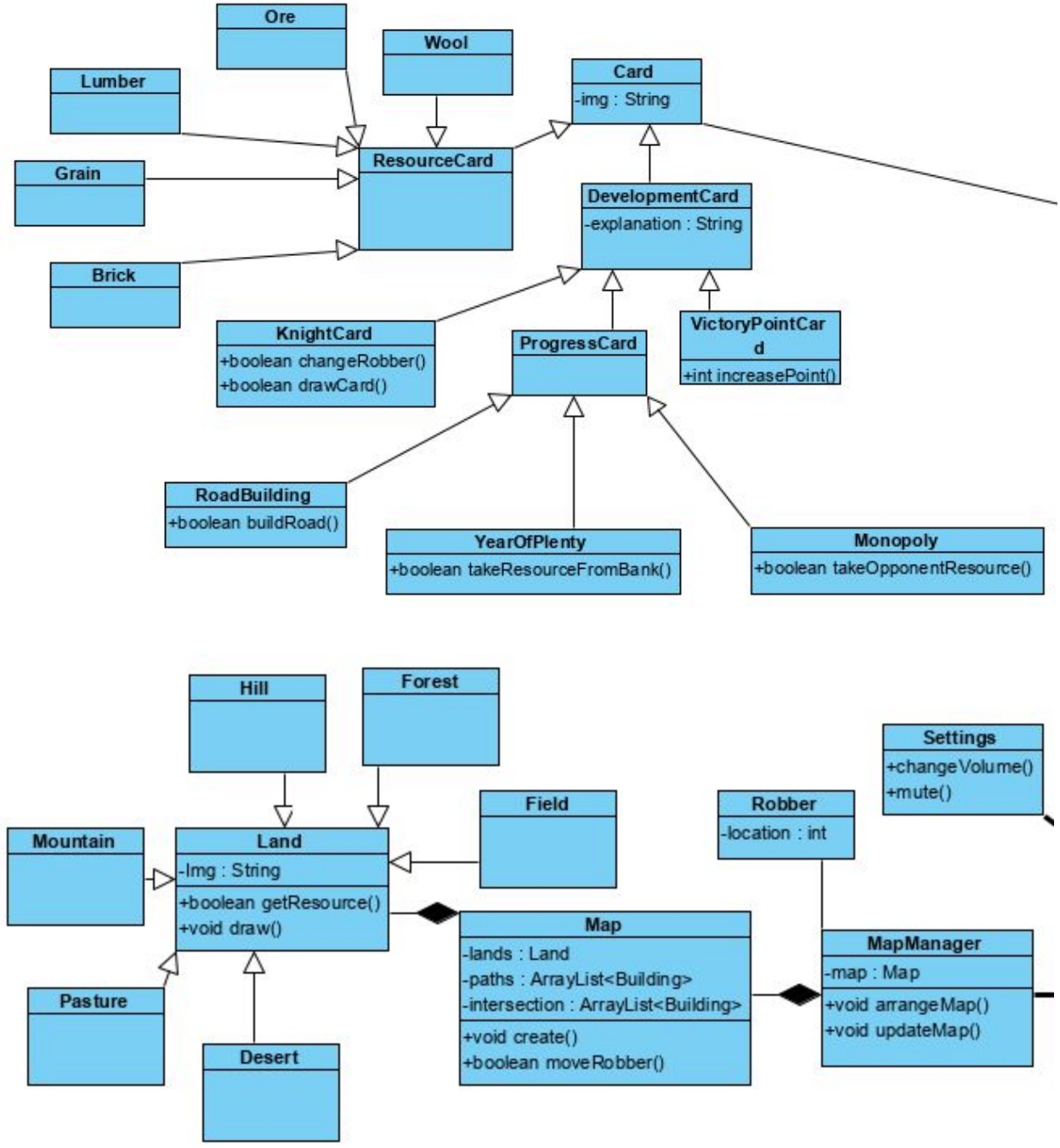

Figure 9 - Second part of Design Model

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. Our sub-systems are models, views and managers.

4.3.1. Game Model

In this area, we have presented our class intercased progressively detailed and clear for any developer to implement it more effectively.

4.3.1.1. Player Class

Attributes:

- **private Building[] building:** This attribute will hold the current buildings.
- **private ResourceCard[] resources:** This attribute will hold the current resources.
- **private int victoryPoint:** This attribute will hold the current victoryPoint.
- **private PrograssCard[] progressCards:** This attribute will hold the current progress cards.
- **private boolean longestRoad:** This attribute will hold if player has the longest road or not.
- **private boolean largestArmy:** This attribute will hold if player has the largest army or not.
- **private string name:** This attribute will hold player's name.

Methods:

- **private boolean playCard():** This method will be used to play card

4.3.1.2. Map Class

Attributes:

- **private Land[] lands:** This attribute will hold the lands on the grid.
- **private ArrayList<Building> paths:** This attribute will hold the buildings built on paths.
- **private ArrayList<Building> intersection:** This attribute will hold the buildings on intersections.

Methods:

- **private void create():** This function creates the game grid at the start of the game. It randomly generates lands and the numbers on it.
- **private boolean moveRobber():** This function moves the robber in the game grid when needed.

4.3.1.3. Card Class

Attributes:

- **private String img:** This attribute is the directory of the image of card.

4.3.1.4. ResourceCard Class

This ResourceCard class is subclass of DevelopmentCard Class

4.3.1.5. Brick Class

This Brick Class is subclass of ResourceCard Class.

4.3.1.6. Grain Class

This Grain Class is subclass of ResourceCard Class.

4.3.1.7. Ore Class

This Ore Class is subclass of ResourceCard Class.

4.3.1.8. Lumber Class

This Lumber Class is subclass of ResourceCard Class.

4.3.1.9. Wool Class

This Wool Class is subclass of ResourceCard Class.

4.3.1.10. DevelopmentCard Class

Attributes:

- **private String explanation:** This attribute will hold explanation of the card.

4.3.1.11. KnightCard Class

Methods:

- **private boolean changeRobber()**: This function moves the location of the knight.

- **private boolean drawCard():** This function makes player to draw a card from a player that has a settlement or a city next to robber's location.

4.3.1.12. VictoryPointCard Class

Methods:

- **private int increasePoint():** This function increase one point of player's points.

4.3.1.13. ProgressCard Class

This Progress Class is subclass of DevelopmentCard Class.

4.3.1.14. RoadBuilding Class

- **private boolean buildRoad():** This function gives extra road to player to build a road to map.

4.3.1.15. YearOfPlenty Class

- **private boolean takeResourceFromBank():** This function add 2 resource cards to player that he chooses.

4.3.1.16. Monopoly Class

- **private boolean takeOpponentResource():** This function steals all resource cards of player's selection from other players.

4.3.1.17. Building Class

Attributes:

- **private int victorPoint:** This attribute will hold the victory points of building.

- **private Player player:** This attribute will hold the which player has the building.

- **private double locationX:** This attribute will hold x location of building.

- **private double locationY:** This attribute will hold y location of building.

4.3.1.18. Settlement Class

This Settlement Class is subclass of Building Class.

4.3.1.19. City Class

This City Class is subclass of Building Class.

4.3.1.20. Road Class

This Road Class is subclass of Building Class.

4.3.1.21. Offer Class

Attributes:

- **private Player sender:** This attribute will hold the offer sender player.
- **private Player receiver:** This attribute will hold the offer receiver player.
- **private ResourceCard offeredItem:** This attribute will hold offered item's resource card.
- **private ResourceCard demandedItem:** This attribute will hold demanded item's resource card.
- **private int offerNum:** This attribute will hold the number of how many resources offered.
- **private int demandNum:** This attribute will hold the number of how many resources demanded.

4.3.1.22 Trade Class

Attributes:

- **private String player:** This attribute will hold the trade offered player.

- **private String tradeItem:** This attribute will hold the offered item.

- **private int tradeRatio:** This attribute will hold the number of given items for the trade.

- **private String demandedItem:** This attribute will hold the demanded item

4.3.1.23. Land Class

Attributes:

- **private String img:** This attribute holds the land images as string on the map like forest, field...etc

Methods:

- **private boolean getResource():** This method takes the resource type and gives the player according to the number on that land.

- **private void draw():** This method draws the lands by using GUI feature of the java according to directory of image directory attribute.

4.3.1.24. Forest Class

This Forest Class is subclass of Land Class.

4.3.1.25. Fields Class

This Fields Class is subclass of Land Class.

4.3.1.26. Hills Class

This Hills Class is subclass of Land Class.

4.3.1.27. Mountains Class

This Mountains Class is subclass of Land Class.

4.3.1.28. Pasture Class

This Pasture Class is subclass of Land Class.

4.3.1.29. Desert Class

This Desert Class is subclass of Land Class.

4.3.2. Views of the game

4.3.2.1. Menu Class

Methods:

- **private void openSettings():** This method will be used for open settings.
- **private void startGame():** This method will be used for start game.
- **private void selectPlayerNum():** This method will be used for selecting player number.

4.3.2.2. Settings Class

Methods:

- **private void mute():** This method will close the voice of the game.
- **private void changeVolume()**: This method will be used for changing volume.

4.3.2.3. PauseMenu Class

Methods:

- **private void openSettings():** This method will be used for open settings.
- **private void mainMenu():** This method will be used for open main menu.

4.3.3. Game Managers

4.3.3.1. GameManager Class

Attributes:

- **private int turnNo:** This attribute holds which player's turn it is.
- **private boolean firstTurn:** This attribute holds if it is the first turn or not.
- **private boolean secondTurn:** This attribute holds if it is the second turn or not.
- **private Card[] initialDevelopmentCardStack:** This attribute holds the randomized stack of DevelopmentCard's.

Methods:

- **private int rollDice():** This function roll the dice at the start of the turn.
- **private boolean distributeResources(int number):** This function distributes the resources of the lands with given number to players.
- **private void startTurn():** This function starts the turn of the next player.
- **private void endTurn():** This function ends the turn of the current player.
- **private Player gameOver():** This function ends the game end returns the winner.

4.3.3.2. PlayerManager Class

Attributes:

- **private Player[] player:** This attribute contains all the player instance of the game. It will be created compositionally when new game is created.

4.3.3.3. MapManager Class

Attributes:

- **private Map map:** This attribute contains the map of the game where map is unique for every game so there will be exactly one map for each game (Singleton) hence map manager contains that unique instance.

Methods:

- **private void arrangeMap():** This method will be used for arranging map.
- **Private void updateMap():** This method will be used for updating map

.

4.3.3.4. BuildManager Class

Attributes:

- **private Buildings[] buildings:** This will contain all the building locations from the start of the game hence to provide information to which location is available or not.

Methods:

- **private boolean buildRoad():** This function will build a road for that player and update the buildings location and players building attributes.

- **private boolean buildCity():** This function will build 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():** This function will build a settlement for that player and update the buildings location, players building attributes and player manager to update the players victory points.

4.3.3.5. TradeManager Class

Methods:

- **private boolean listTrades():** This function will list all the possible trades and check their ratio from that player's building locations (harbours).

- **private boolean tradeResources():** This method gives the player a chance to trade the same type of resource cards with a card that the player chooses. The ratio of the trade changes according to the players location of the map. As an example, if the player has a harbour, he can trade his resource cards in 3:1 ratio with the bank.

- **private boolean buyProgressCards():** This method trades the resource cards of a player with a random progress card.

4.3.3.6. OfferManager Class

Attributes:

- **private Offer[] offers:** This attribute holds all the user offers inside an array of offer objects.

Methods:

- **private void makeOffer():** This function will create new offer and stack it in offers array where the correspondent of the will be able to view it.

- **private void listOffers():** This function will list the relevant offers for that player from all offer array.

- **private boolean findOffer():** This function will find the relevant offer for that player from all offer array.

- **private void closeOffer():** This function will close all the offers for that user in that turn.

- **private boolean acceptOffer():** This function will accept that offer instance and inform player managers player array.

- **private boolean declineOffer():** This function will decline that offer and delete that offer instance from offer array.

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.