Bilkent University

# CS 319 Term Project

## Design Report

**Team Name: Group 2H CA**

**Project Name: Settlers of Catan**

**Members:**

• Ravan Aliyev

• Berrak Taşkınsu

• Hazal Aksu

• Ayşe Nursu Savaşkan

# Table of Contents

# 1. Introduction

In this section purpose of the system and our main design goals are explained.

## 1.1. Purpose of the System

Settlers of Catan is a multiplayer board game. The purpose of our program is to create the digital version of an already existing physical game and to make the game more attractive by adding different features and creating a new mode to satisfy the users. In addition to the original version of the game, there is a new mode which is single player mode where user plays against the Artificial Intelligence. This game is planned to be platform-independent software, which allows this game to be played on Linux, Mac OS and Microsoft Windows operating systems. This software will be user-friendly, which allows the users to be able to learn how to play this game easily. This competitive game will challenge the players and entertain them by improving their analything thinking abilities and spatial reasoning aptitude. The main goal is getting ten points before other players by building cities, settlements and roads and by the help of the development cards.

## 1.2. Design Goals

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.[1] These requirements are listed in functional requirements sections in the analysis report of this software and these requirements will be discussed below. Design goals are qualities or properties that software has a solution for these properties. They are found by analyzing the requirements and the information that we have about the game and will give us a direction while developing the software. Significant design goals for the digital version of our game are listed below:

***Performance Goals:*** Response time of the software will not take time more than 200 ms. Free space in memory for the software will be less than 0.4 GB .

***Maintenance Goals:*** The game should be implemented as a multi-platform software, which allows it to be played on Windows, MacOS and Linux operating systems in which JRE 8 is supported. Extensibility is also an important software designing criteria for maintenance, which we should pay attention while developing the software. This criteria increases the ability of the software for further extension and improvements. It also allows other developers to add new functions or modify the existing software too and this extensibility criteria provides for modifications without impairing the existing software. Project related documents will be open source too, and this will help other developers to redesign our system. To make it easy to understand our software by reading the codes, we have to pay attention to another maintenance criteria which is readability.

***Dependability Goals:*** Simplicity of our system allows it not to have any security risks, because it does not contain any personal information. Simplicity of our system lets it to have a high reliability.

***End User Goals:*** Usability is a criteria that shows the ease of use and learnability by users. Our system can be used by every user that has an ability to use a computer to achieve objectives in game with effectiveness, efficiency and satisfaction.

# 2. High-level Software Architecture

## 2.1. Subsystem Decomposition (Packages)

There are two packages in our software: GUI Package and Game Control Package. The GUI Package contains the classes subject to the user interface and the Game Control Package has the classes that deal with non-GUI elements of our game objects. Several GUI classes are dependent on Game Control classes as shown in the figure below.

Most of the interactions between the two packages occur through the Controller and Player classes.

GamePanel class of the GUI Package is dependent on GameBoard and LandHexes classes of the Game Control Package to check/get information about the placement of previous settlements on the board for the player to be able to place/upgrade settlements on the board.

Most moves made in the game are dealt by the Player class of the Game Control Package. Thus, several GUI classes that have buttons to do those moves (OfferTradePanel, TradePanel, GamePanel, BoardPanel) are dependent on the Player class.

GUI Package classes MainMenuPanel and StartGamePanel are dependent on Controller class of Game Control Package to be able to start the game.

GUI class BankPanel is dependent on Game Control Package's Controller class to be able to pick a random card and is dependent on the Bank class to be able to trade cards with the bank in the game.

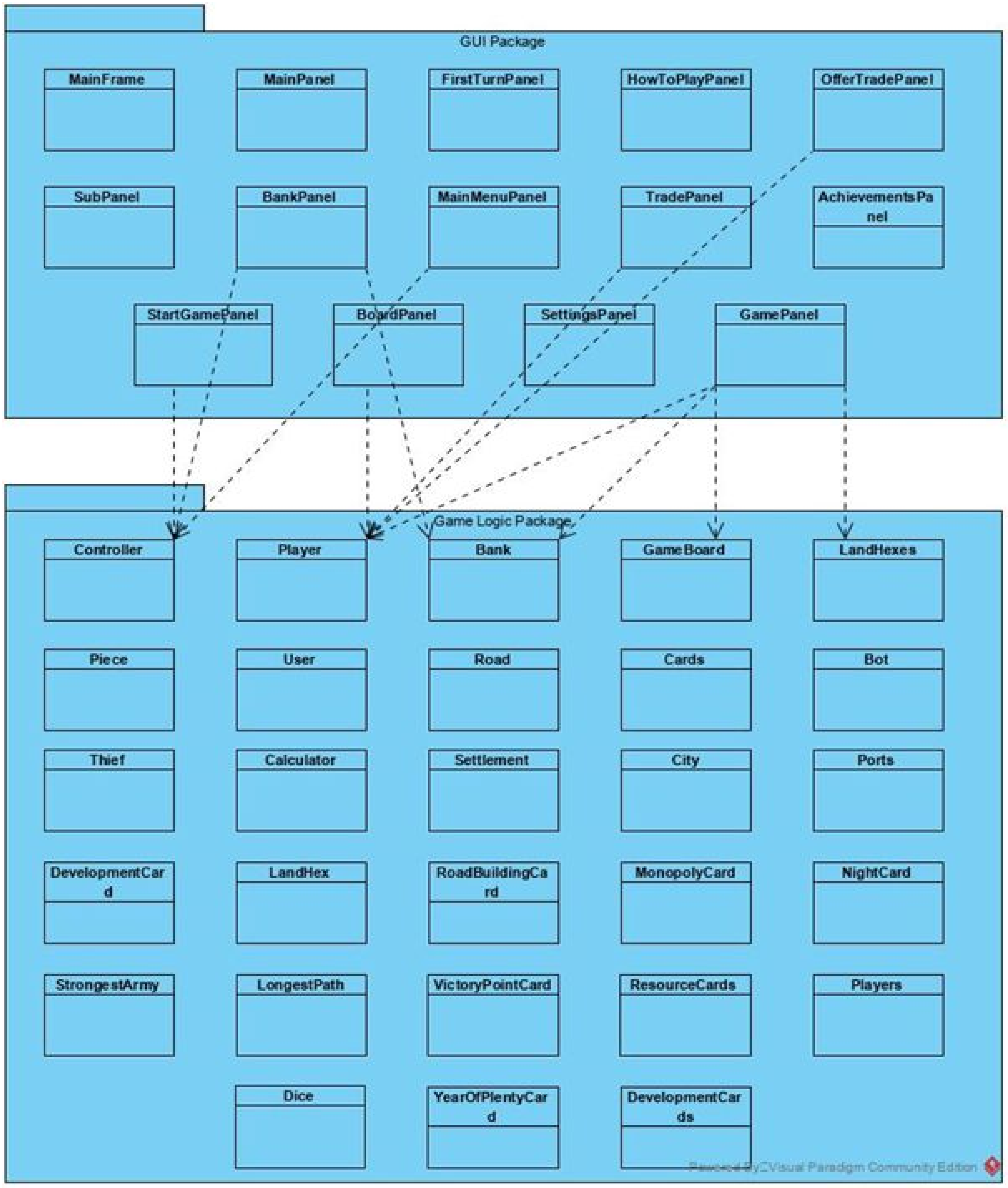


*Figure 1: Packages*

## 2.2. Hardware/Software Mapping

For the implementation of our software we do not use any external components, so there is not any distinction between user machines and server. That is why we do not do any hardware/software mapping.

## 2.3. Persistent Data Management

Only 3 data need to be stored for the Settlers of Catan game: saving settings, achievements and highscores. Our software does not need to save complex data since it is a simple game. Achievements and highscores will be defined according to the players' progresses. Those progresses and settings can be saved as a combination of strings and integers arrays. Because of the simplicity of this software, it does not require access to other applications and so, no application does need to be able to access data. Since this small amount of information needs to be stored, we will not use high level of data storage. Considering the points listed above, we think that using database to store the data of the game is not necessary and will be excessive. So, using flat files instead of database will allow our program to work much faster.

## 2.4. Access Control and Security

Subsystems of our software is shown at section 2.1 by explaining how and why they are chosen in such a way. Our game does not need to have access to the internet, which makes it secure, because it is not open to dangers coming from the internet. Single player mode will be played against Artificial Intelligence. Even multiplayer mode does not need internet access to the internet because it will be played on users' local devices. According to the system design of our software, “Credits” and “How To Play” will be accessible to all users and users will not be able to modify these information. Only “Setting” and progresses like “Achievements” and “HighScores” will be specific to every user. But after the game ends “Achievements” and “HighScores” will be checked by the “Controller” subsystems whether player's desires are appropriate to execute.

## 2.5. Boundary Conditions

This software does not need to be installed which means that it has not any executable .exe extension file. The game will be executed from the .jar file, enabling it to be portable among different operating systems and devices. Since it will be executed from .jar file internet connection is not needed to open the game. Playing the game on the local device and no need for the internet ensures that there will be no network or connection problem in the game. Even though there can be other failures too. For example, user terminates the game which can cause loss of the current game situation. This can cause potential data loss. Another problem can occur while reading files. Problem which occurs while reading files can cause to the game start without a sound or images.  This problem can be fixed by modifying the corrupted file.

# 3. Subsystem Services

## 3.1. Menu Subsystem

The UML Class Diagram for the GUI Package is provided below.

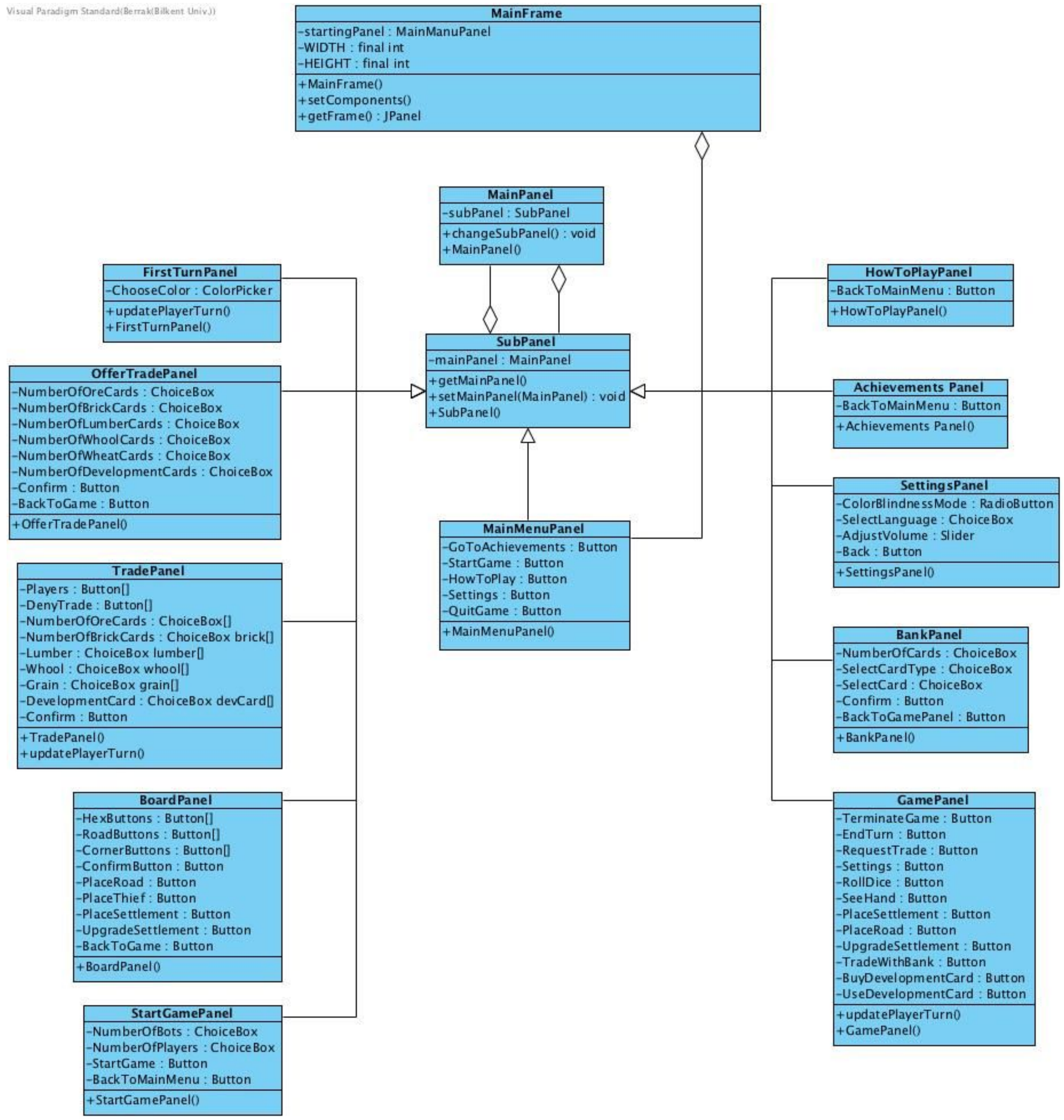


*Figure 2: GUI Package*

## 3.1.1. MainFrame

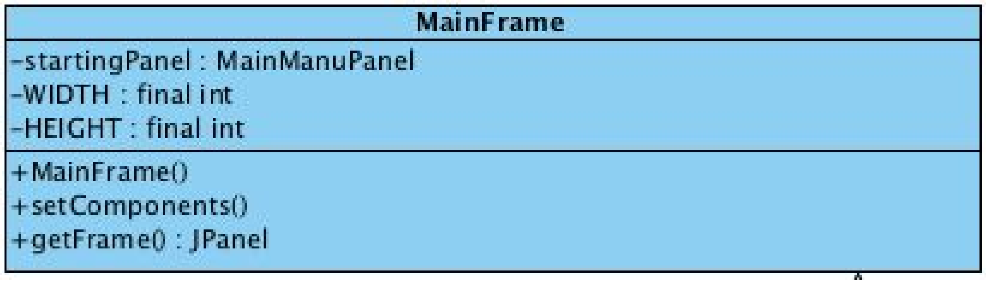


*Figure 3: MainFrame*

MainFrame is a JFrame which we use as the main window of our application.

## 3.1.14. MainPanel

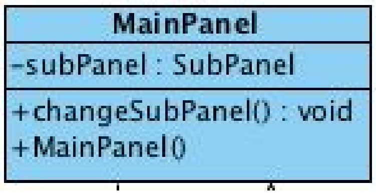


*Figure 4: MainPanel*

**Attributes:**

- private SubPanel subPanel: stores the current SubPanel that is shown to the user.
- private String Language: stores the current language of the game. Initially English.
- private bool ColorBlingnessMode: stores true when colorBlindnessMode is on. Initially off.

**Constructors:**

- public MainPanel( ): constructor of the MainPanel. Default language is English and color blindness mode is off.

**Methods:**

- public void changeSubPanel( ): changes the current SubPanel that is shown to the user.

- public void setColorBlindnessMode( boolean colorBlindnessMode ): sets the ColorBlindnessMode attribute to the parameter colorBlindnessMode.

- Public void setLanguage( String language ): sets the Language attribute to the parameter language.

### 3.1.13. SubPanel

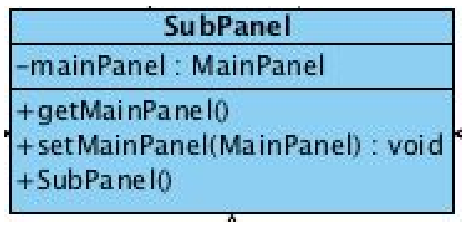


*Figure 5: SubPanel*

**Attributes:**

- private MainPanel mainPanel: stores the MainPanel for the purpose of accessing it and allowing MainPanel to change the current SubPanel.

- private String Language: stores the current language of the game.

- private bool ColorBlingnessMode: stores true when colorBlindnessMode is on.

**Constructors:**

- public SubPanel( ): constructor of the SubPanel. By default, the language is English and the color blindness mode is off.

- public SubPanel( String language, boolean colorBlindnessMode): constructor of the SubPanel. Sets the Language attribute of the SubPanel to the parameter language and the ColorBlindnessMode attribute to the parameter colorBlindnessMode.

**Methods:**

- public MainPanel getMainPanel(): accessor of the MainPanel attribute.

- public void setMainPanel(): mutator of the MainPanel attribute.

## 3.1.2. MainMenuPanel

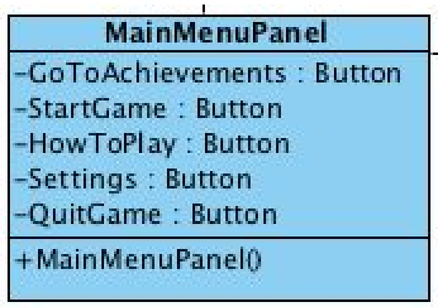


*Figure 6: MainMenuPanel*

**Attributes:**

- private Button GoToAchievements: allows the user to go to the AchievementsPanel. When clicked, MainMenuPanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into an AchievementsPanel.

- private Button StartGame: allows the user to go to the StartGamePanel. When clicked, MainMenuPanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a StartGamePanel.

- private Button HowToPlay: allows the user to go to the HowToPlayPanel. When clicked, MainMenuPanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into an HowToPlayPanel.

- private Button Settings: allows the user to go to the SettingsPanel. When clicked, MainMenuPanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a SettingsPanel.

- private Button QuitGame: a Button object that allows the user to terminate the game. On mouse click, MainFrame will be closed.

### 3.1.3. HowToPlayPanel

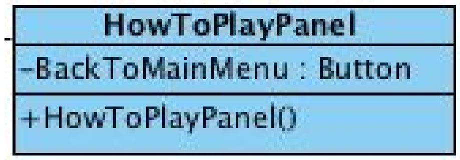


*Figure 7: HowToPlayPanel*

**Attributes:**

- private Button BackToMainMenu: allows the user to go to the MainMenuPanel. When clicked, HowToPlayPanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a MainMenuPanel.

### 3.1.4. AchievementsPanel

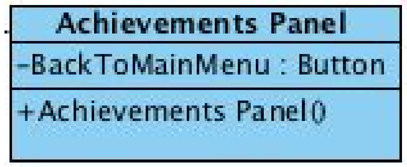


*Figure 8: AchievementsPanel*

**Attributes:**

- private Button BackToMainMenu: allows the user to go to the MainMenuPanel. When clicked, AchievementsPanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a MainMenuPanel.

### 3.1.5. SettingsPanel

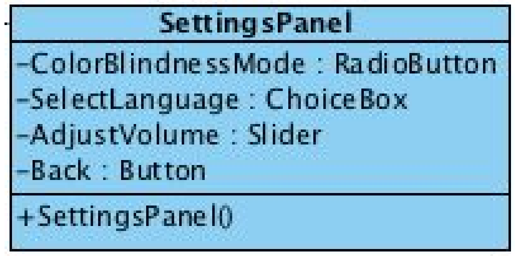


*Figure 9: SettingsPanel*

**Attributes:**

- private String PreviousPanel: holds the SubPanel which called the SettingsPanel. This will either be a MainMenuPanel or a GamePanel.

- private Button Back: allows the user to go to the previous SubPanel, which can either be a MainMenuPanel or a GamePanel. When clicked, SettingsPanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel depending on the attribute PreviousPanel.

- private RadioButton ColorBlindnessMode: allows the user to switch to the color blindness mode.

- private ChoiceBox SelectLanguage: allows the user to select between English, Turkish, Azerbaijani and Russian.

- private Slider AdjustVolume: allows the user to set the volume of the game.

- private Button ApplyChanges: changes the Language and ColorBlindnessMode attributes of the MainPanel.

**Constructors:**

- public SettingsPanel( SubPanel previousPanel ): constructor of the SettingsPanel. It takes a single parameter previousPanel and modifies the PreviousPanel attribute accordingly.

### 3.1.6. FirstTurnPanel

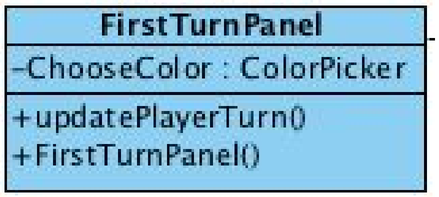


*Figure 10: FirstTurnPanel*

**Attributes:**

- private ColorPicker ChooseColor: allows the user to pick a color for the Player.

### 3.1.7. StartGamePanel

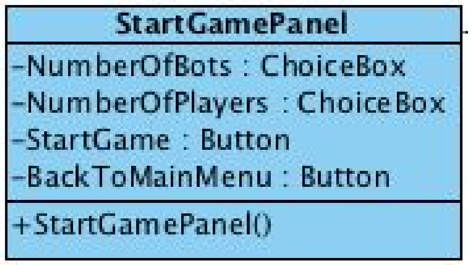


*Figure 11: StartGamePanel*

**Attributes:**

- private ChoiceBox NumberOfBots: allows the user to determine the number of bots.*
- private ChoiceBox NumberOfPlayers: allows the user to determine the number of human players.*
- private Button StartGame: allows the user to go to the GamePanel. When clicked, StartGamePanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a GamePanel.
- private Button BackToMainMenu: allows the user to go to the MainMenuPanel. When clicked, StartGamePanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a MainMenuPanel.

*The game allows 3 or 4 players, therefore the user must make a selection accordingly.

### 3.1.8. GamePanel

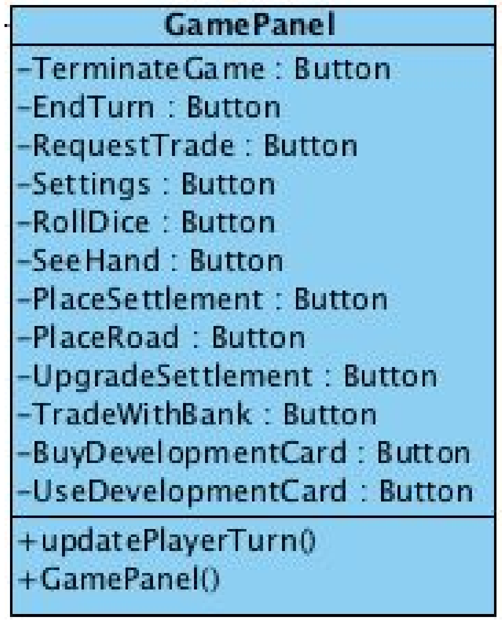


*Figure 12: GamePanel*

**Attributes:**

- private Button TerminateGame: allows the user to quit the game while playing. All progress for achievements is lost and the user is directed to the MainMenuPanel.

- private Button EndTurn: When clicked, the next player who has the turn is highlighted and all functionality for the current user ends.

- private Button RequestTrade: allows the user to make a trade request via going to the OfferTradePanel. When clicked, GamePanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a OfferTradePanel.

- private Button Settings: allows the user to open the in game settings via going to the SettingPanel. When clicked, GamePanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a SettingsPanel.

- private Button RollDice: allows the player to roll the dice.

- <u>private Button SeeHand:</u> allows the user to view his/her hand. A smaller panel will be opened at the bottom of the GamePanel that displays the current Player’s hand. All functionality of the GamePanel will still be active to the user.

- <u>private Button PlaceSettlement:</u> allows the user to place a settlement via going to the BoardPanel. When clicked, GamePanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a BoardPanel.

- <u>private Button PlaceRoad:</u> allows the user to place a road via going to the BoardPanel. When clicked, GamePanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a BoardPanel.

- <u>private Button UpgradeSettlement:</u> allows the user to upgrade his/her settlement via going to the BoardPanel. When clicked, GamePanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a BoardPanel.

- <u>private Button TradeWithBank:</u> allows the user to make a trade with the bank via going to the BankPanel. When clicked, GamePanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a BoardPanel.

- <u>private Button BuyDevelopmentCard:</u> calculates the current Player’s resources. If enough resources are present, the player buys a development card.

- <u>private Button UseDevelopmentCard:</u> allows the Player to use the development card. Visible only if the a development card is selected.

## 3.1.9. BankPanel

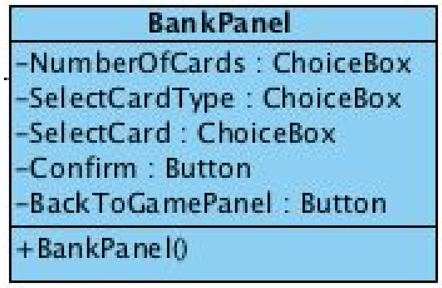

BankPanel
-NumberOfCards : ChoiceBox
-SelectCardType : ChoiceBox
-SelectCard : ChoiceBox
-Confirm : Button
-BackToGamePanel : Button
+BankPanel()

*Figure 13: BankPanel*

**Attributes:**

- private ChoiceBox NumberOfCards: allows user to select the number of cards of the specified type to trade with the desired card.

- private ChoiceBox SelectCardType: allows user to select the card type to give.

- private ChoiceBox SelectCard: allows user to select the desired card.

- private Button Confirm: allows user to confirm the selection. When clicked, BankPanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a TradePanel.

- private Button BackToGamePanel: allows the user to go back to the Game Panel. When clicked, BankPanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a GamePanel.

## 3.1.10. BoardPanel

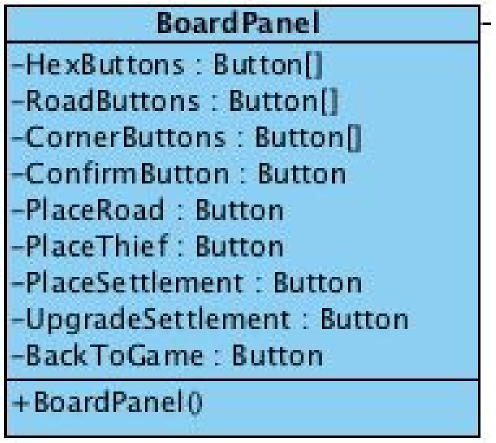


*Figure 14: BoardPanel*

**Attributes:**

- private Button HexButtons[ ]: a list of Buttons for every hex on the board. When clicked, the selected hex will be highlighted.

- private Button PathButtons[ ]: a list of Buttons for every path, that  is for every side of every hex. When clicked, the selected path will be highlighted.

- private Button CornerButtons[ ]: a list of Buttons for every corner of hexes on the board. When clicked, the selected corner will be highlighted.

- private Button ConfirmButton: allows the user to confirm the changes made. When clicked, When clicked, the game will be updated and the highlighted selections will be processed.

- private Button PlaceRoad: allows the user to place a road on the board.

- private Button PlaceThief: allows the user to replace the thief. Only visible when the current player rolls 7 or plays a monopoly card.

- private Button PlaceSettlement: allows the user to place settlements.

- private Button UpgradeSettlement: allows the user to upgrade current settlements.

- private Button BackToGame: allows the user to exit the BoardPanel. When clicked, BoardPanel will access the MainPanel via the getMainPanel() method it inherited from the abstract class SubPanel and the changeSubPanel() method of the MainPanel will be called to change the current panel into a GamePanel.

### 3.1.11. OfferTradePanel

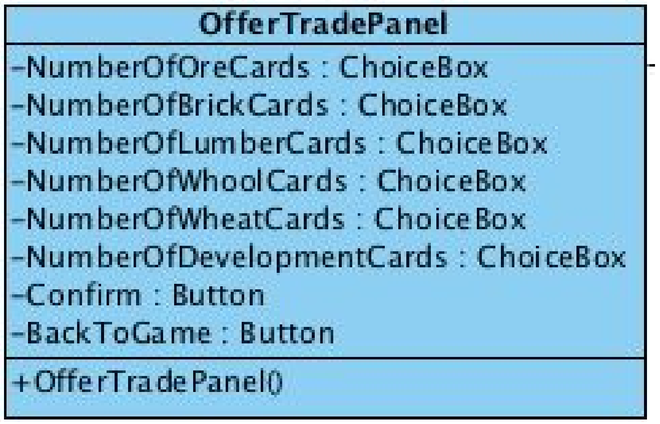

OfferTradePanel
-NumberOfOreCards : ChoiceBox
-NumberOfBrickCards : ChoiceBox
-NumberOfLumberCards : ChoiceBox
-NumberOfWhoolCards : ChoiceBox
-NumberOfWheatCards : ChoiceBox
-NumberOfDevelopmentCards : ChoiceBox
-Confirm : Button
-BackToGame : Button
+OfferTradePanel()

*Figure 15: OfferTradePanel*

**Attributes:**

- private ChoiceBox NumberOfOreCards: allows the user to select the desired number of ore cards.

- private ChoiceBox NumberOfBrickCards: allows the user to select the desired number of brick cards.

- private ChoiceBox NumberOfLumberCards: allows the user to select the desired number of lumber cards.

- private ChoiceBox NumberOfWoolCards: allows the user to select the desired number of wool cards.

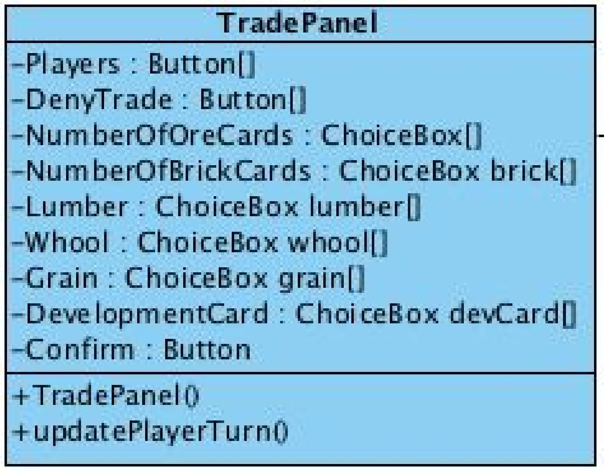
- private ChoiceBox NumberOfWheatCards: allows the user to select the desired number of wheat cards.

- private ChoiceBox NumberOfDevelopmentCards: allows the user to select the desired number of development cards.

- private Button Confirm: allows the user to confirm the trade request in terms of the desired cards. When clicked, OfferTradePanel will access the MainPanel via the getMainPanel( ) method it inherited from the abstract class SubPanel and the changeSubPanel( ) method of the MainPanel will be called to change the current panel into a TradePanel.

- private Button BackToGame: allows the user to discard trade request. When clicked, OfferTradePanel will access the MainPanel via the getMainPanel( ) method it inherited from the abstract class SubPanel and the changeSubPanel( ) method of the MainPanel will be called to change the current panel into a GamePanel.

### 3.1.12. TradePanel



TradePanel
-Players : Button[]
-DenyTrade : Button[]
-NumberOfOreCards : ChoiceBox[]
-NumberOfBrickCards : ChoiceBox brick[]
-Lumber : ChoiceBox lumber[]
-Whool : ChoiceBox whool[]
-Grain : ChoiceBox grain[]
-DevelopmentCard : ChoiceBox devCard[]
-Confirm : Button
+TradePanel()
+updatePlayerTurn()

*Figure 16: TradePanel*

**Attributes:**

- private Button Players[ ]: a list to store all Button's for all players except for the player in turn. When clicked, the player will be selected by the user in turn.

- private Button DenyTrade[ ]: a list to store all Button's for all players except for the player in turn. Available for a player if it is the trading turn of that player

- private ChoiceBox NumberOfOreCards[ ]: a list of all ChoiceBox'es to chose the number of desired ore cards, for all players except the one in turn. Available for a player if it is the trading turn of that player and the player has accepted the trade offer.

- private ChoiceBox NumberOfBrickCards[ ]: a list of all ChoiceBox'es to chose the number of desired brick cards, for all players except the one in turn. Available for a player if it is the trading turn of that player and the player has accepted the trade offer.

- <u>private ChoiceBox NumberOfLumberCards[ ]:</u> a list of all ChoiceBox'es to chose the number of desired lumber cards, for all players except the one in turn. Available for a player if it is the trading turn of that player and the player has accepted the trade offer.

- <u>private ChoiceBox NumberOfWoolCards[ ]:</u> a list of all ChoiceBox'es to chose the number of desired wool cards, for all players except the one in turn. Available for a player if it is the trading turn of that player and the player has accepted the trade offer.

- <u>private ChoiceBox NumberOfWheatCards[ ]:</u> a list of all ChoiceBox'es to chose the number of desired wheat cards, for all players except the one in turn. Available for a player if it is the trading turn of that player and the player has accepted the trade offer.

- <u>private ChoiceBox NumberOfDevelopmentCards[ ]:</u> a list of all ChoiceBox'es to chose the number of desired development cards, for all players except the one in turn. Available for a player if it is the trading turn of that player and the player has accepted the trade offer.

- <u>private Button Confirm:</u> allows a player to confirm what resources he/she wants in exchange for what the player in turn wants. When clicked, space allocated for the player will be highlighted in green, all functionality of the player will be lost and the trading turn will move on to the next player.

## 3.2. Game Logic Subsystem

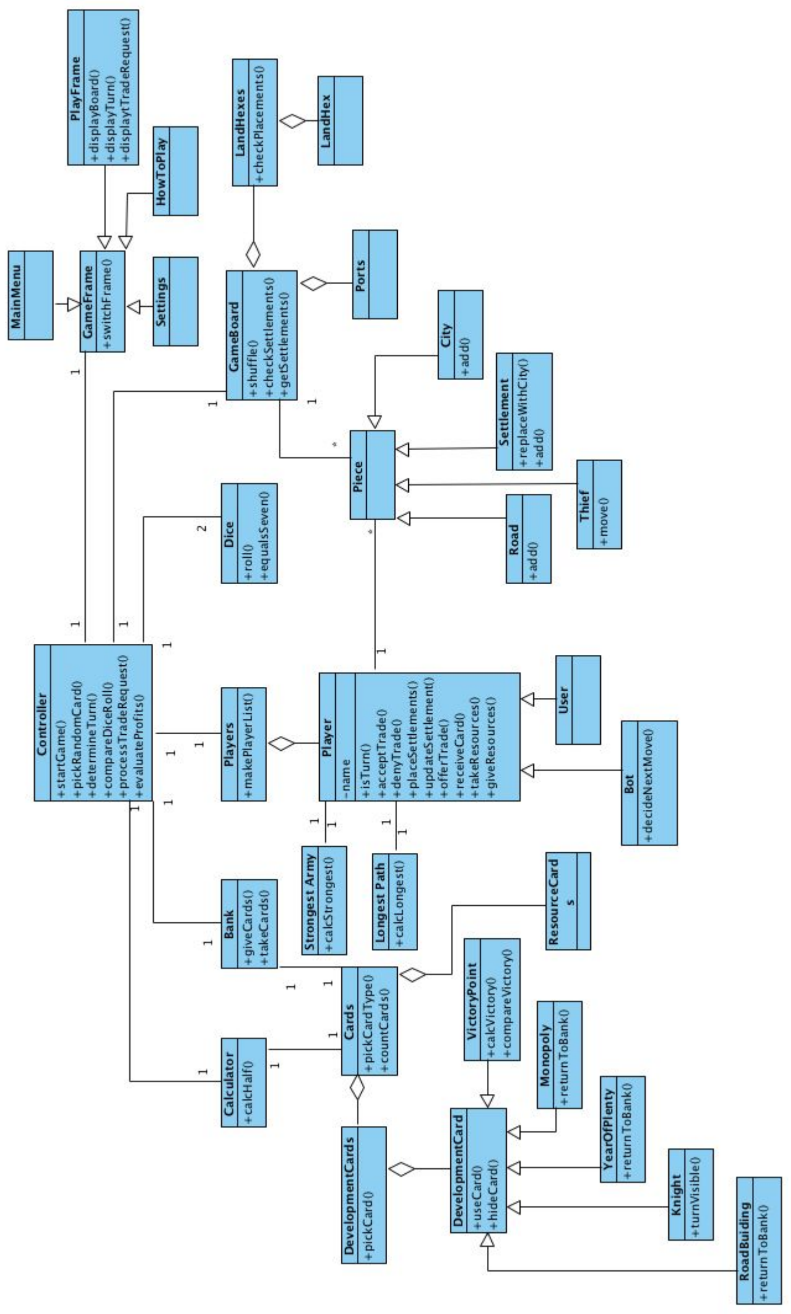


*Figure 17: GameLogicPackage*

## 3.1.1. Controller

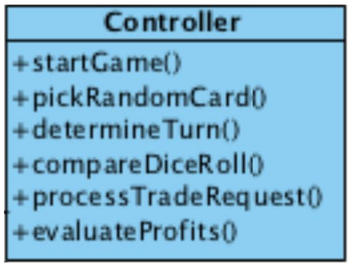


*Figure 18: Controller*

**Methods:**

- startGame(): starts the game for desired number of players and bots.
- pickRandomCard(): picks a random card and gives it to the player in turn from the selected player's hand.
- determineTurn(): determine player turn
- compareDiceRoll(): for the first turn only, calculated greatest dice roll to determine the first player.
- processTradeRequest(): processes trade requests.
- evaluateProfits(): evaluate how many points a player gets at the end of a turn

## 3.2.2. Players

Players
+makePlayerList()

*Figure 19: Players*

**Methods:**

- makePlayerList(): create a player list

## 3.2.3. Bank

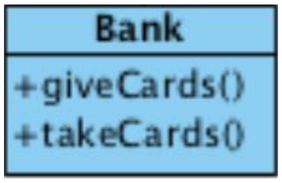


*Figure 20: Bank*

**Methods:**

- giveCards(): give the selected cards to the selected Player.
- takeCards(): take selected cards from the selected Player.

## 3.2.4. Player

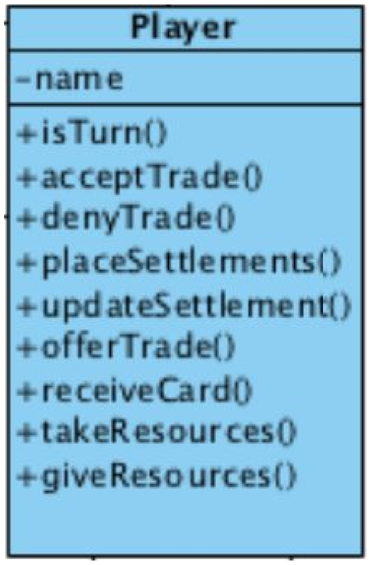


*Figure 21: Player*

**Attributes:**

- private String name: stores the names of a player

**Methods:**

- public boolean isTurn(): calculates whether if it is this Player's turn.

- public void acceptTrade(): accepts the trade request.
- public void denyTrade(): denies the trade request.
- public void placeSettlements(): moves the desired number of settlements from the hand of the player and places them on the selected corners of the board.
- public void updateSettlements(): upgrades the selected settlements into cities.
- public void offerTrade(): processes offer trade.
- public void receiveCard():  receive resources and add them to players hand.
- public void takeResources(): receive resources from the bank and add them to players hand.
- public void giveResources(): discard resources from hand.

### 3.2.5. Dice

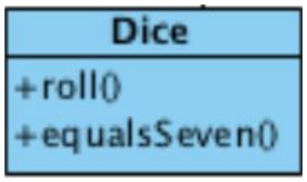


*Figure 22: Dice*

**Methods:**

- public int roll(): randomly calculates a dice roll and returns the value.
- public boolean equalsSeven(): determines if the current dice roll equals 7.

### 3.2.6. GameBoard

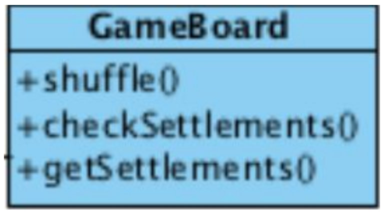


*Figure 23: GameBoard*

**Methods:**

- public void shuffle(): shuffles the hexes and the numbers and places and creates and array.
- public boolean checkSettlements(): calculate whether the settlements are correct.
- public void getSettlements(): receive settlements from the Controller

### 3.2.8. Calculator

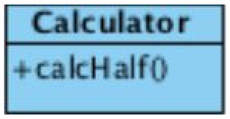


*Figure 24: Calculator*

**Methods:**

- public calcHalf(): Calculates half of the cards in a player's hand.

### 3.2.9. Cards

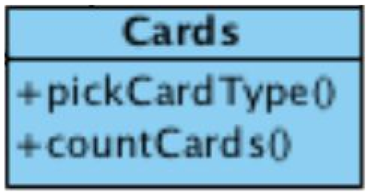


*Figure 25: Cards*

**Methods:**

- public pickCardType(): select a card type.
- public countCards(): count the cards in the array.

### 3.2.10. DevelopmentCards

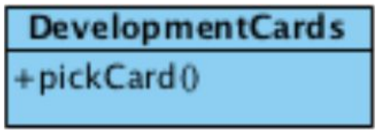


*Figure 26: DevelopmentCards*

**Methods:**

- public pickCard(): pick a random card.

### 3.2.11. DevelopmentCard

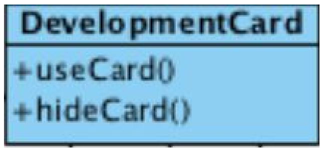


*Figure 27: DevelopmentCard*

**Methods:**

- public void useCard(): gets the card from the player's hand and sends it to the board.
- public void hideCard(): hides the cards in players hand if it is not their turn.

### 3.2.12. RoadBuildingCard

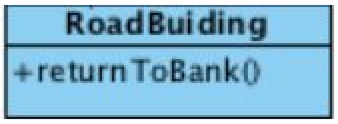


*Figure 28: RoadBuildingCard*

**Methods:**

- public returnToBank(): add this card to the bank.

### 3.2.13. KnightCard

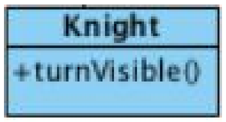


*Figure 29: KnightCard*

**Methods:**

- public turnVisible(): turn visible when on board.

### 3.2.14. YearOfPlentyCard

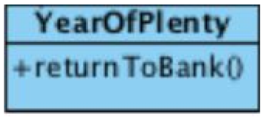


*Figure 30: YearOfPlentyCard*

**Methods:**
- public returnToBank(): add this card to the bank

### 3.2.15. MonopolyCard

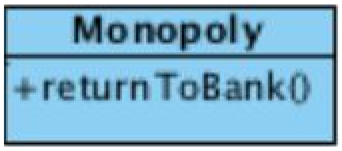


*Figure 31: MonopolyCard*

**Methods:**
- public returnToBank(): add this card to the bank

## 3.2.16. VictoryPointCard

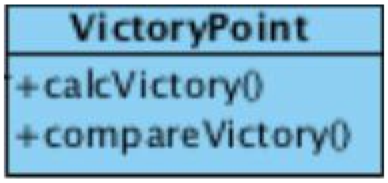


*Figure 32: VictoryPointCard*

**Methods:**

- public calcVictory(): calculate victory points of a player
- public compareVictory(): compare the victory points of players to determine the greatest victory points.

## 3.2.18. StrongestArmy

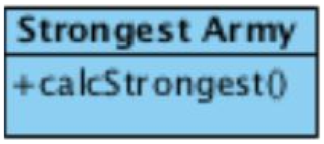


*Figure 33: StrongestArmy*

**Methods:**

- public calcStrongest(): calculate the strongest army.

## 3.2.19. LongestPath

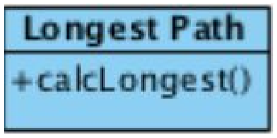


*Figure 34: LongestPath*

**Methods:**

- public calcLongest(): calculate the longest path.

### 3.2.20. Bot

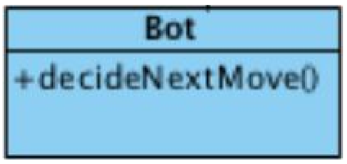


*Figure 35: Bot*

**Methods:**

- <u>public decideNextMove():</u> decide on the next move if it is this player's turn.

### 3.2.22. Road

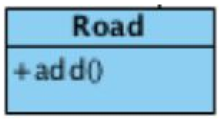


*Figure 36: Road*

**Methods:**

- <u>public void add():</u> adds itself to the game board, to the desired path.

### 3.2.23. Thief

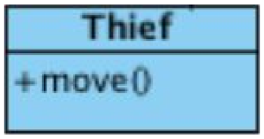


*Figure 37: Thief*

**Methods:**

- <u>public void move():</u> move to the desired hex.

### 3.2.24. Settlement

*Figure 38: Settlement*

**Methods:**

- public replaceWithCity(): replace this settlement with a city.
- public add(): add a settlement to the board.

### 3.2.25. City

*Figure 39: City*

**Methods:**

- public add(): places a city by replacing the selected settlement.

### 3.2.27. LandHexes

*Figure 40: LandHexes*

**Methods:**

- public checkPlacements(): determines which players will get which and how many resources from the land hexes.

# 4. Low-level Design

Low-Level design of project will be described in this part.

## 4.1 Object Design Trade-Offs

It is not possible to perform some design goals at the same time. So, we have to choose which one of them to be prioritized over another. Our choices for design goals are listed below

### *Functionality vs Usability*

In our game, we decided to prioritize usability over functionality. Since controls of our game is easy and it is a fun based game, usability is more important for our software. The game is controlled by mouse and this makes it easy to understand. “How To Play” screen increases our usability of our software.

### *Efficiency vs Portability*

We chose portability over efficiency. Because we want to make our game enable to everyone we will design a platform independent software. Our software will be written in Java platform and it will be runnable on every operating systems and devices that supports JRE 8.

### *Security vs Usability*

Simplicity of our game lets us not to be concerned about the security issues. We use just file system to store the needed data and we do not have to use database. Players do not need to login by using user/password system in our design of the game, because we prioritize usability and do not limit the user for security purposes.

## 4.2. Final Object Design

Final object design is provided below as a UML Class Diagram.

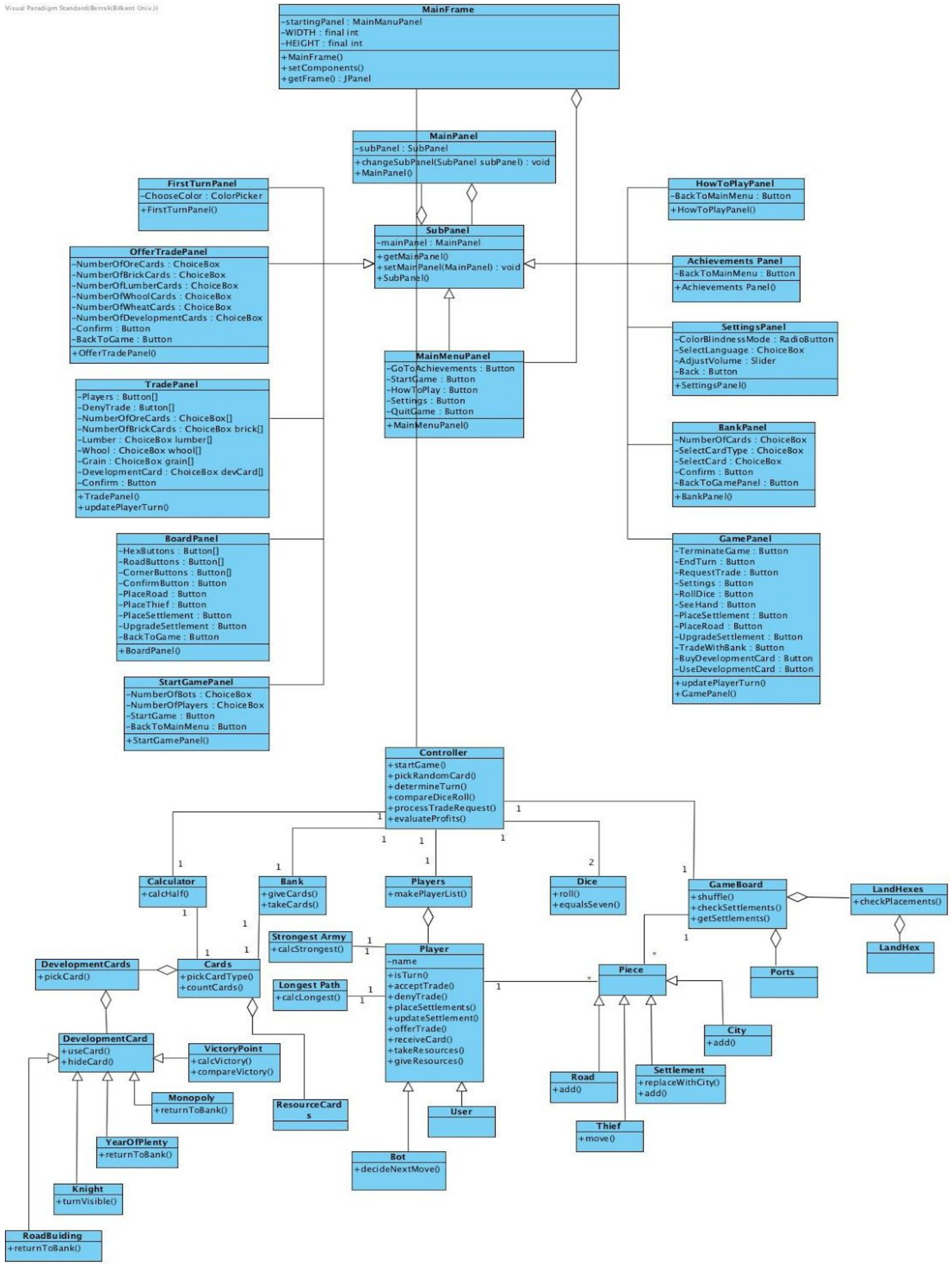


*Figure 41: Final Object Design Diagram*

## 4.3. Packages

We will be using Java to write our software. We will use Java's Standard Library for the back-end part of our project and JavaFX to implement the GUI part. The packages that will be used are provided below.

Package java.fx.*: For the user interfaces, we will use java.fx.

Package java.io.*: Provides for system input and output through data streams, serialization and the file system.

Package java.util.*:Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

## 4.4. Class Interface

Our software does not use any class interfaces.

# 5. Glossary & References

[1] https://www.tutorialspoint.com/software_engineering/software_design_basics.htm

[2] http://tutorials.jenkov.com/javafx/index.html

[3]https://docs.oracle.com/javase/7/docs/api/index.html