Bilkent University

Department of Computer Engineering

Object-Oriented Software Engineering Group Project

CS 319 Project: Monopoly Bidding War - Group 1D

Design Report

Deniz Kasap 21702460, Irmak Çeliker 21702502, Batuhan Budak 21704212,

Kübra Okumuş 21600980, Ömer Faruk Kayar 21602452

Instructor: Eray Tüzün

Teaching Assistant(s): Emre Sülün, Barış Ardıç and Elgun Jabrayilzade

Progress/Design Report Iteration 1

Nov 29, 2020

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Object-Oriented Software Engineering course CS319.

Contents

1. Introduction

1.1 Purpose of the System

Monopoly Bidding War is a board game that will be offered on a digital platform. Rules are the same with the original Monopoly with a slight difference, every player will be given the chance to attend the auction for any estate and in any round. The winner will be the person with higher wealth, so the players will buy estates, negotiate with each other, and build houses to bankrupt other players. The game is designed to eliminate the complexity for the users along with to give aesthetic satisfaction. The system also offers functional requirements such as, how to play, settings, etc., that also aims to provide convenience for the users.

1.2 Design Goals

The aim of the game's design is to have high in-game and server performance, to be user-friendly and to be extendable. To do so, the plot of the game and the interfaces are designed to be more fluent and convenient for the users, its graphic designs are elaborate and also offers players many different alternatives.

1.2.1 Reliability and Efficiency

A database is used to interact and exchange data between users. The data flow between the database and the computer is very constricted, so there will be no loss of information. Host's computers will distribute the crucial and needed information through the database. This feature will increase the efficiency of the game by making it faster and more fluent.

1.2.2 User-Friendliness

The game is designed in a sensible way so that players will not come across with complexities. Taking turns and operation orders are done less complexly, in order to maintain a user-friendly design. Also, the objects and the details in the game have smooth graphic designs. By the help of soft drawings and chirpy colors, it is aimed to bring aesthetic satisfaction to the users. Most of the buttons and control units are eliminated to sustain simplicity and to avoid confusion for the users that creates a user-friendly environment.

1.2.3 High Performance

The pawns are demonstrated as 3D objects and move on the map. This feature offers a high performance experience to the users while the aim is to keep the players engaged to the game and to bring an aesthetic view. Also, players will be able to send each other trade requests that will allow players a chance to maintain a relationship with the other players. All of the operations are designed in a way to keep players interested and engaged with the game in order to give the best experience for them and to maintain a high performance.

1.2.4 Flexibility

There are many different squares and operations offered in the game such as auction, trading, veto rights, paying taxes, etc. This variance provides flexibility for the users since they are given the chance to perform different operations throughout the game. The aim behind this goal is to expand choices and make the system engaging so that players will feel satisfied with the game.

1.2.5 Ease of learning

The system will offer a How to Play option for the users in order to maintain simplicity and understandability for the users. Even though the game offers different operations such as trading, auctioning, veto rights, mortgaging, etc, to maintain better experience, by the help of the how to play option players can adapt and understand the game very easily.

2. High-level Software Architecture

2.1 Subsystem Decomposition

We decided to use three layer Architectural Style in our project. Our architecture consists of a presentation layer, application layer and data layer. Presentation layer is the user interface part of the game which provides communication between the user and the second layer. Application layer is the second layer which is also described as the Game Control subsystem. It consists of Game Square, Game, Player, Assets_Package and Storage packages. Storage can be seen as a data access layer as well, it maintains the data flow between the database and the Game Control subsystem. Lastly, the third layer is the Data Layer named as Data Management subsystem. This layer contributes the data to all corresponding users.

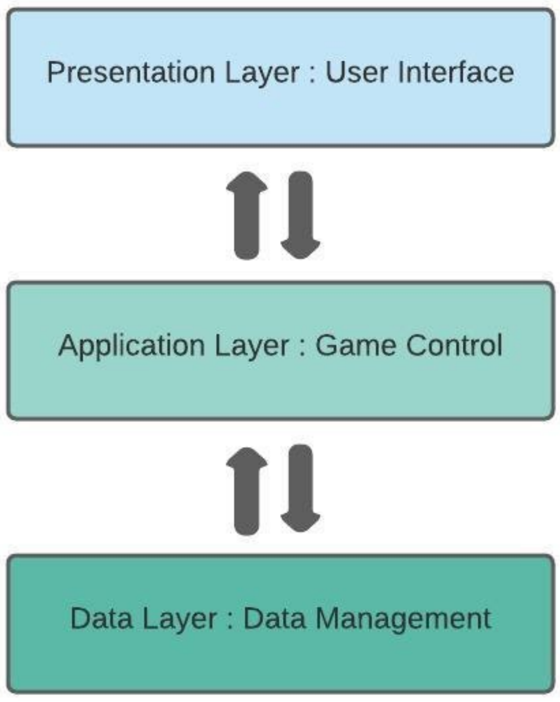
Presentation Layer : User Interface
Application Layer : Game Control
Data Layer : Data Management

Figure 1: 3-Layer Architecture

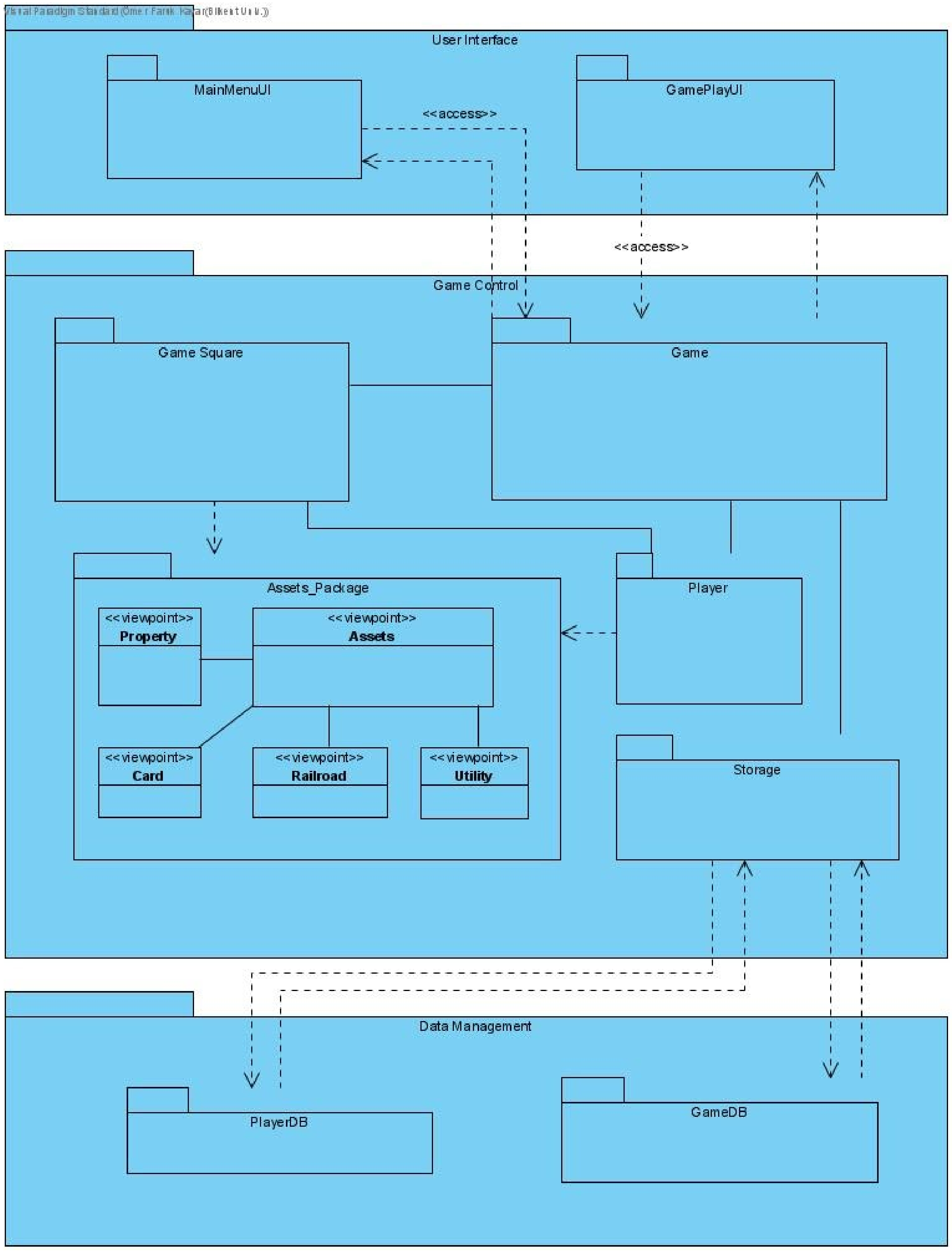

Figure 2: Subsystem Decomposition

2.2 Hardware/Software Mapping

Our game doesn't require any hardware specification since it is a simple board game with database compatibility. For server specifications, it only requires an internet connection. For hardware specifications, It will only require up to 10MB available space in storage. Default hardware setup would manage the game without any additional build. Besides, in order to play the game, the one will need a computer, a monitor. We used the latest version of the Microsoft Visual Studio Runtime Environment which will be running on Windows. We used C# language to code our game.

2.3 Persistent Data Management

We will be using Firebase Real Time Database to store the data and distribute only the required data to all players in the game. Our main goal is to carry the least amount of information possible in order to maintain data flow faster. In order to make it, we decided to use a local distributor class named storage.

The storage class will take game board information and player information from the host. However, storage classes of non-host players will not take any information related to their game board from their system, it will be received from the host via database. In order to maintain our goal, the database will be checked by storage when a player makes a decision on the game. For example, when a bid routine has ended, the results which are provided by the database will be displayed.

There will be a gameboard database class which has dice result, current playing player id, list of assets on the board, remaining chance and community chest cards, and building datas. There will be player database classes which have player id, list of assets, keepable card information, and money amount.

2.4 Access Control and Security

Only the host actor can create the ‘GameSquare’. After the creation of the ‘GameSquare’ host actor has the same accessibility of an ordinary player. Once the ‘GameSquare’ is created, ‘Game’ initializes assets and creates a storage. In game routines such as taking a turn, bidding and trading are in control of the ‘Game’, the host and other player(s) don't have access to these event’s orders. However, the host and other player(s) can access and manipulate their own assets and events only in their turns. Exit option is always accessible to all players. The host and the player cannot directly access the database, however the ‘Game’ updates the changes made in the database instead of them. This data is reflected by the ‘Game’ to all other players via the user interface. Database will not be a public server, this will ensure the security of the data. Also, this is an online game without any save feature. This makes the server security easier.

	GameSquare	Assets	Storage

Host(Player)	<<create>>
	view()
	takeTurn()
	trade()
	veto()
	exit()i
		<<create>>
		view()
		select()
		buy()
			<<create>>
			update()
			getValue()

Player	view()
	takeTurn()
	trade()
	veto()
	exit()
		<<create>>
		view()
		select()
		buy()
			getValue()

Figure 3: Access Diagram

2.5 Boundary Conditions

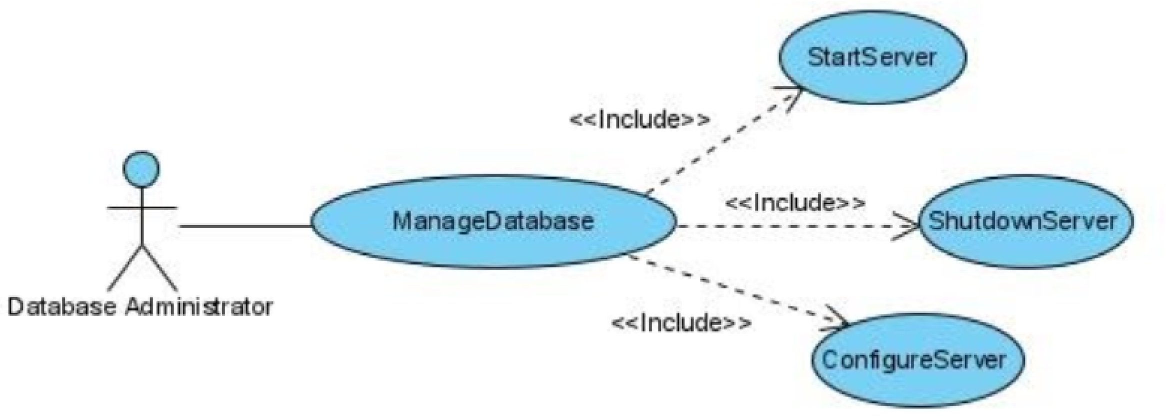

Figure 4: Boundary Use Cases

2.5.1 Initialization

In order to start the application, the game doesn't require anything except visual studio runtime environment. While hosting a game, game id should be unique. Otherwise, the application will terminate a message and it will proceed to the main menu. If the game id is a unique id, a file containing storage data will be created, and a utilizable space in the database will be created. While joining a game it will be checked if there is a corresponding server in the database. An error message will appear if there is no such game in the server and the application will proceed to the main menu. If there is a suitable server, a file containing the player's information will be created in the database and in the storage.

2.5.2 Termination

After the game finishes, all storage data and database data will be deleted. We don’t provide a save feature since it is an online game.

2.5.3 Failure

When the system confronts an error, the last operation will be executed only one more time. If it doesn’t solve the problem, it will proceed to ErrorOccured. ErrorOccured use case will handle the error. If it is an server related error, data will be protected and server will be restarted. If it is a game related error, the game will be stopped. In future implementations, we will be receiving error reports to a specified email.

3. Low-level Design

3.1 Object Design Trade-offs

3.1.1 Quality vs Functionality

While working on this project, a decision we took regarding our direction was to follow a path similar to Hasbro's. We did not want to define the whole genre from scratch, but rather wanted to create a fun, balanced and familiar monopoly game, which is different from the others, but still familiar enough. As a result of this decision, to create a familiar, balanced, enjoyable but still different enough game, we choose to introduce only a couple of new functionalities to the game. We added a bid and a veto option. Apart from our new functionalities, our approach was to use the fact that we have a computer-system as a platform, and use this platform to enhance the already existing features of a game. For example, the bidding in our game is done anonymously, only the computer knows all of the bids, and asks the person who landed on the square to bid again in the case of him not winning that bidding war. This functionality is a hard/impossible feature to add to a board game. However, to assure balance, quality and a sense of familiarity. We needed to opt out of many useful, fun or creative features.

3.1.2 Processing Power & Memory vs Readability & Maintainability

Since our project is relatively small, easy to build and light-weight for a computer these days, we decided to have well defined objects with a lot of variables clearly defined with their functionalities. We could have tried to write this game as efficiently as possible, with very little memory allocations, with as few objects as possible. But since this is a group project, communication is the hardest challenge to overcome. We decided instead of having a code segment that is short & efficient, but really hard to understand and develop further, an easily understandable code with a little more memory allocations was a better approach.

3.1.3 Development Time vs User Experience

During the analysis of the “Monopoly Bidding War” game, we have decided to implement interface components and graphics with Visual Studio since Visual Studio supports C# we decided to code the game with C#. Also during the analysis, we decided to implement an isometric game map to increase the user’s experience. To implement the game map, the player pawns in a better way, we have decided to use Guna 2 Framework which provides better-looking buttons, labels, and transparent picture boxes. By choosing implementation with Visual Studio, development time will be easier for us, and with the benefits of Guna 2 Framework, we will have better graphics and user experience.

3.2 Final Object Design

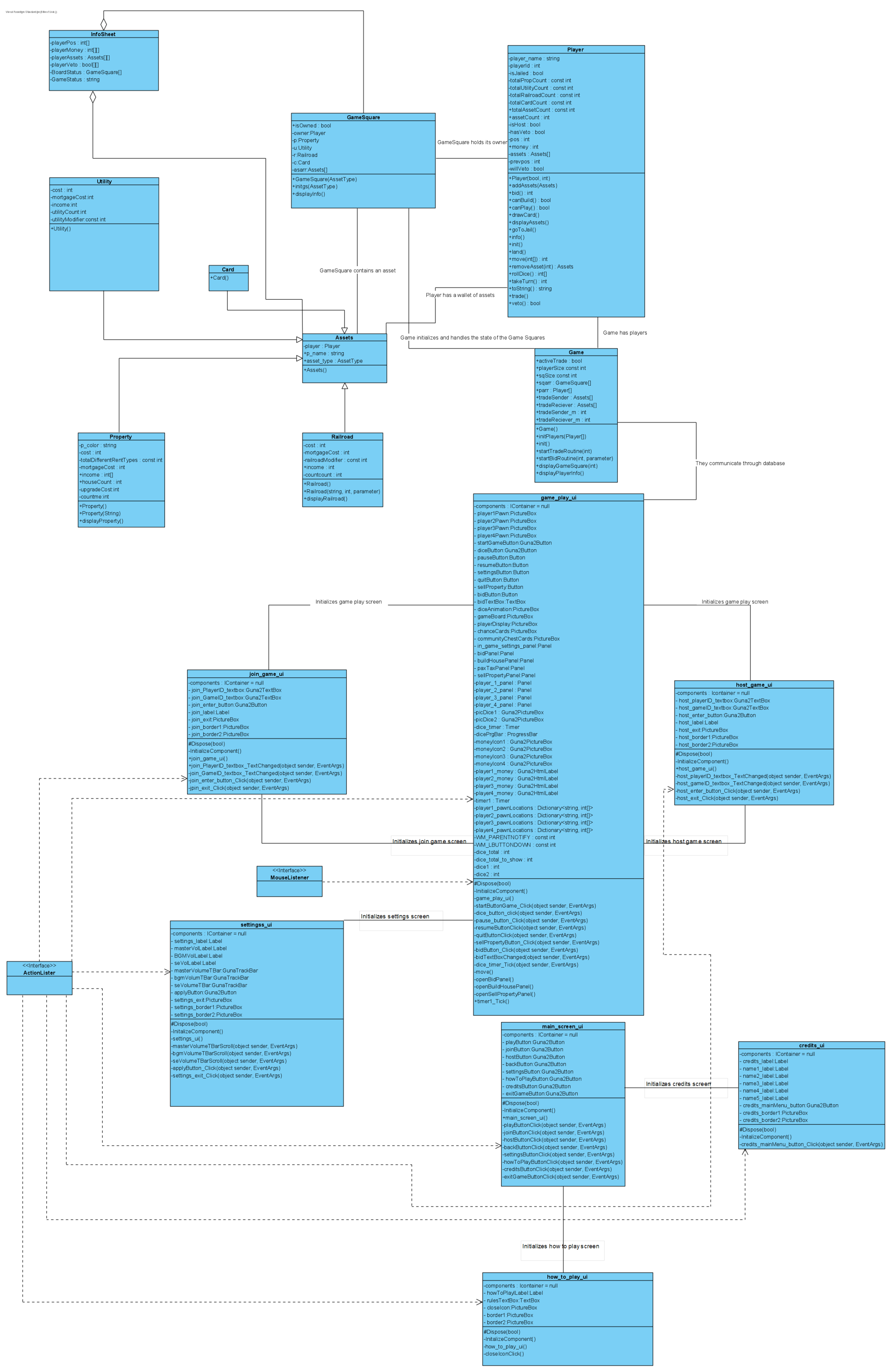
InfoSheet
-playerPos : int[]
-playerMoney : int[][]
-playerAssets : Assets[][]
-playerVeto : bool[][]
-BoardStatus : GameSquare[]
-GameStatus : string
GameSquare
+isOwned : bool
-owner:Player
-p:Property
-u:Utility
-r:Railroad
-c:Card
-asarr:Assets[]
+GameSquare(AssetType)
+initgs(AssetType)
+displayInfo()
Player
-player_name : string
-playerId : int
-isJailed : bool
-totalPropCount : const int
-totalUtilityCount : const int
-totalRailroadCount : const int
-totalCardCount : const int
+totalAssetCount : const int
+assetCount : int
-isHost : bool
-hasVeto : bool
-pos : int
+money : int
-assets : Assets[]
-prevpos : int
-willVeto : bool
+Player(bool, int)
+addAssets(Assets)
+bid() : int
+canBuild() : bool
+canPlay() : bool
+drawCard()
+displayAssets()
+goToJail()
+info()
+init()
+land()
+move(int[]) : int
+removeAsset(int) : Assets
+rollDice() : int[]
+takeTurn() : int
+toString() : string
+trade()
+veto() : bool
GameSquare holds its owner
Utility
-cost : int
-mortgageCost:int
-income:int
-utilityCount:int
-utilityModifier:const int
+Utility()
Card
+Card()
GameSquare contains an asset
Player has a wallet of assets
Assets
-player : Player
+p_name : string
+asset_type : AssetType
+Assets()
Game initializes and handles the state of the Game Squares
Game has players
Game
+activeTrade : bool
+playerSize:const int
+sqSize:const int
+sqarr : GameSquare[]
+parr : Player[]
+tradeSender : Assets[]
+tradeReciever : Assets[]
+tradeSender_m : int
+tradeReciever_m : int
+Game()
+initPlayers(Player[])
+init()
+startTradeRoutine(int)
+startBidRoutine(int, parameter)
+displayGameSquare(int)
+displayPlayerInfo()
Property
-p_color : string
-cost : int
-totalDifferentRentTypes : const int
-mortgageCost : int
+income : int[]
+houseCount : int
-upgradeCost:int
-countme:int
+Property()
+Property(String)
+displayProperty()
Railroad
-cost : int
-mortgageCost : int
-railroadModifier : const int
+income : int
-countcount : int
+Railroad()
+Railroad(string, int, parameter)
+displayRailroad()
They communicate through database
game_play_ui
-components : IContainer = null
- player1Pawn:PictureBox
- player2Pawn:PictureBox
- player3Pawn:PictureBox
- player4Pawn:PictureBox
- startGameButton:Guna2Button
- diceButton:Guna2Button
- pauseButton:Button
- resumeButton:Button
- settingsButton:Button
- quitButton:Button
- sellProperty:Button
- bidButton:Button
- bidTextBox:TextBox
- diceAnimation:PictureBox
- gameBoard:PictureBox
- playerDisplay:PictureBox
- chanceCards:PictureBox
- communityChestCards:PictureBox
- in_game_settings_panel:Panel
- bidPanel:Panel
- buildHousePanel:Panel
- paxTaxPanel:Panel
- sellPropertyPanel:Panel
-player_1_panel : Panel
-player_2_panel : Panel
-player_3_panel : Panel
-player_4_panel : Panel
-picDice1 : Guna2PictureBox
-picDice2 : Guna2PictureBox
-dice_timer : Timer
-dicePrgBar : ProgressBar
-moneyIcon1 : Guna2PictureBox
-moneyIcon2 : Guna2PictureBox
-moneyIcon3 : Guna2PictureBox
-moneyIcon4 : Guna2PictureBox
-player1_money : Guna2HtmlLabel
-player2_money : Guna2HtmlLabel
-player3_money : Guna2HtmlLabel
-player4_money : Guna2HtmlLabel
-timer1 : Timer
-player1_pawnLocations : Dictionary<string, int[]>
-player2_pawnLocations : Dictionary<string, int[]>
-player3_pawnLocations : Dictionary<string, int[]>
-player4_pawnLocations : Dictionary<string, int[]>
-WM_PARENTNOTIFY : const int
-WM_LBUTTONDOWN : const int
-dice_total : int
-dice_total_to_show : int
-dice1 : int
-dice2 : int
#Dispose(bool)
-InitializeComponent()
-game_play_ui()
-startButtonGame_Click(object sender, EventArgs)
-dice_button_click(object sender, EventArgs)
-pause_button_Click(object sender, EventArgs)
-resumeButtonClick(object sender, EventArgs)
-quitButtonClick(object sender, EventArgs)
-sellPropertyButton_Click(object sender, EventArgs)
-bidButton_Click(object sender, EventArgs)
-bidTextBoxChanged(object sender, EventArgs)
-dice_timer_Tick(object sender, EventArgs)
-move()
-openBidPanel()
-openBuildHousePanel()
-openSellPropertyPanel()
+timer1_Tick()
Initializes game play screen
Initializes game play screen
join_game_ui
-components : IContainer = null
- join_PlayerID_textbox:Guna2TextBox
- join_GameID_textbox:Guna2TextBox
- join_enter_button:Guna2Button
- join_label:Label
- join_exit:PictureBox
- join_border1:PictureBox
- join_border2:PictureBox
#Dispose(bool)
-InitializeComponent()
+join_game_ui()
+join_PlayerID_textbox_TextChanged(object sender, EventArgs)
-join_GameID_textbox_TextChanged(object sender, EventArgs)
-join_enter_button_Click(object sender, EventArgs)
-jpin_exit_Click(object sender, EventArgs)
host_game_ui
-components : Icontainer = null
- host_playerID_textbox:Guna2TextBox
- host_gameID_textbox:Guna2TextBox
- host_enter_button:Guna2Button
- host_label:Label
- host_exit:PictureBox
- host_border1:PictureBox
- host_border2:PictureBox
#Dispose(bool)
-InitializeComponent()
+host_game_ui()
-host_playerID_textbox_TextChanged(object sender, EventArgs)
-host_gameID_textbox_TextChanged(object sender, EventArgs)
-host_enter_button_Click(object sender, EventArgs)
-host_exit_Click(object sender, EventArgs)
Initializes join game screen
Initializes host game screen
<<Interface>>
MouseListener
Initializes settings screen
settingss_ui
-components : IContainer = null
- settings_label:Label
- masterVolLabel:Label
- BGMVolLabel:Label
- seVolLabel:Label
- masterVolumeTBar:GunaTrackBar
- bgmVolumTBar:GunaTrackBar
- seVolumeTBar:GunaTrackBar
- applyButton:Guna2Button
- settings_exit:PictureBox
- settings_border1:PictureBox
- settings_border2:PictureBox
#Dispose(bool)
-InitalizeComponent()
-settings_ui()
-masterVolumeTBarScroll(object sender, EventArgs)
-bgmVolumeTBarScroll(object sender, EventArgs)
-seVolumeTBarScroll(object sender, EventArgs)
-applyButton_Click(object sender, EventArgs)
-settings_exit_Click(object sender, EventArgs)
<<Interface>>
ActionLister
main_screen_ui
-components : IContainer = null
- playButton:Guna2Button
- joinButton:Guna2Button
- hostButton:Guna2Button
- backButton:Guna2Button
- settingsButton:Guna2Button
- howToPlayButton:Guna2Button
- creditsButton:Guna2Button
- exitGameButton:Guna2Button
#Dispose(bool)
-InitializeComponent()
+main_screen_ui()
-playButtonClick(object sender, EventArgs)
-joinButtonClick(object sender, EventArgs)
-hostButtonClick(object sender, EventArgs)
-backButtonClick(object sender, EventArgs)
-settingsButtonClick(object sender, EventArgs)
-howToPlayButtonClick(object sender, EventArgs)
-creditsButtonClick(object sender, EventArgs)
-exitGameButtonClick(object sender, EventArgs)
Initializes credits screen
credits_ui
-components : IContainer = null
- credits_label:Label
- name1_label:Label
- name2_label:Label
- name3_label:Label
- name4_label:Label
- name5_label:Label
- credits_mainMenu_button:Guna2Button
- credits_border1:PictureBox
- credits_border2:PictureBox
#Dispose(bool)
-InitalizeComponent()
-credits_mainMenu_button_Click(object sender, EventArgs)
Initializes how to play screen
how_to_play_ui
-components : Icontainer = null
- howToPlaylLabel:Label
- rulesTextBox:TextBox
- closeIcon:PictureBox
- border1:PictureBox
- border2:PictureBox
#Dispose(bool)
-InitalizeComponent()
-how_to_play_ui()
-closeIconClick()

Figure 5: Class Diagram

3.3 Packages

3.3.1 System.Drawing

This package provides access to GDI+ basic graphics functionality. More advanced functionality is provided in the System.Drawing.Drawing2D, System.Drawing.Imaging, and System.Drawing.Text namespaces.

3.3.2 System.Text

This package contains classes that represent ASCII and Unicode character encodings; abstract base classes for converting blocks of characters to and from blocks of bytes; and a helper class that manipulates and formats String objects without creating intermediate instances of String.

3.3.3System.Windows.Forms

This package contains classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system.

3.3.4 System.Component.Model

This package provides classes that are used to implement the run-time and design-time behavior of components and controls.

3.3.5 System.Collections.Generic

This package contains interfaces and classes that define generic collections, which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.

3.3.6 System.Data

This package provides access to classes that represent the ADO.NET architecture. ADO.NET lets you build components that efficiently manage data from multiple data sources.

3.3.7 System.Threading.Tasks

This package Provides types that simplify the work of writing concurrent and asynchronous code. The main types are Task which represents an asynchronous operation that can be waited on and cancelled, and Task<TResult>, which is a task that can return a value. The TaskFactory class provides static methods for creating and starting tasks, and the TaskScheduler class provides the default thread scheduling infrastructure.

3.4 Class Interface

3.4.1 Action Listener

This interface will be invoked when an action occurs and it will receive action events. We will implement this interface on game_play_ui, join_game_ui, host_game_ui, settings_ui, main_screen_ui and credits_ui classes. It will give action info to these classes to track action events.

3.4.2 MouseListener

This interface will be invoked whenever a mouse action is received from the user. This interface will track mouse locations. We will implement this interface on the game_play_ui class and game_play_ui class will perform the required operations depending on this interface.

4 Glossary & References

[1] Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010, ISBN-10: 0136066836.

[2] https://www.hasbro.com/common/instruct/monins.pdf

[3] Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and Iterative Development, by Craig Larman, Prentice Hall, 2004, ISBN: 0-13-148906-2.

[4] Microsoft Docs

https://docs.microsoft.com/en-us/dotnet/