



Group 12

Final Report

CS 319 - 01

“ μ Monopoly”

Authors:

Alper Önder

Ani Kristo

Buse Burcu Akçadağ

Ali Göçer

Contents

1.	Introduction.....	3
2.	Requirement Analysis	3
2.1.	Overview	3
2.2.	Functional Requirements.....	4
2.3.	Non-functional Requirements	6
2.3.1.	Usability requirements	6
2.3.2.	Performance requirements.....	6
2.3.3.	Supportability requirements	7
2.4.	Constraints (Pseudo-requirements).....	7
2.4.1.	Implementation requirements.....	7
2.4.2.	Packaging requirements	7
2.4.3.	Legal requirements.....	7
2.5.	Scenarios.....	7
2.6.	Use Case Models.....	9
2.7.	User Interface	13
3.	Analysis	18
3.1.	Object Model.....	18
3.1.1.	Domain Lexicon	18
3.1.2.	Class Diagram	20
3.2.	Dynamic Models.....	21
3.2.1.	State Chart.....	21
3.2.2.	Sequence Diagrams	22
4.	Design	26
4.1.	Design Goals.....	26
4.2.	Sub-System Decomposition	27
4.2.1.	Graphical User Interface Sub-system.....	28
4.2.2.	Controllers Sub-system	29
4.2.3.	Models Sub-system	29
4.3.	Architectural Patterns	30
4.3.1.	Model-View-Controller (MVC).....	30
4.3.2.	Three-Tier Architectural Style	31

4.4.	Hardware / Software Mapping	31
4.5.	Addressing Key Concerns.....	32
4.5.1.	Persistent Data Management	32
4.5.2.	Access Control and Security	33
4.5.3.	Global Software Control	33
4.5.4.	Boundary Conditions	33
5.	Object Design.....	35
5.1.	Pattern Application	35
5.1.1.	Abstract Factory Pattern.....	35
5.1.2.	Builder Pattern.....	36
5.1.3.	Observer Pattern	37
5.1.4.	Façade Pattern	37
5.1.5.	Composite Pattern.....	38
5.2.	Class Interfaces.....	38
5.3.	Specifying Contracts	55
6.	Conclusion	60

1. Introduction

μMonopoly is a smaller scale software implementation of the famous Monopoly board game. The Monopoly game is a simulation of entrepreneurship: there are 2-4 players who are assigned a certain capital and they need to invest it on property or buildings in order to collect rent from the other players and increase their income. The players who run out of the financial capital and assets are said to be bankrupt and they lose the game, thus, the remaining last player will be declared as winner. During the game there are some twists to make it more challenging and interesting by the action of the bank – a standalone, abstract entity which controls the game.

As this is part of the Object Oriented Software Engineering course, we aim at providing a completely OOP implementation, and this report will reflect the design for this software. As part of the Analysis, we have mentioned the Functional and Nonfunctional requirements, the Constraints and the System Models which include the Use Case models, the Object models, the Dynamic models and finally the User Interface mockup. Moreover, we continue with our Design, thus including the Design Goals, Sub-System Decomposition, Hardware / Software Mapping, Architectural Patterns, Persistent Data Management, Access Control, Global Software Control Flow and Boundary Conditions. The next section will lead to a brief explanation of the project as part of the overview.

2. Requirement Analysis

2.1. Overview

We decided on working on a desktop based, Java application which will simulate the famous board game of Monopoly. The basic rules are:

- It is a multiplayer game, where each player can only play in turn, not simultaneously.
- The users have a token and a certain amount of money.
 - The user is able to purchase a token upgrade such as a gold token or platinum token. Gold tokens decrease the amount of rent which is needed to pay to the competitor by 10% and the platinum token decreases the amount of money the user needs to pay in order to purchase the land, build houses and build hotels by 10%.
- The users roll two dice and move on the board according to the sum of the dice faces. In case they roll doubles they have the chance to roll again. Three consecutive doubles send the user to Jail.
 - The user is able to purchase a dice upgrade such as gold or platinum dice. Gold dice make the user roll twice and platinum dice make the user move two times the sum of the roll.
- In order to go out of the Jail the users have to roll doubles the next turn, wait for three turns or use a Get-Out-Of-Jail card.
- As the users move along the squares they can either buy a new square (land) if it is not taken otherwise pay the rent to the owning user.
- The amount of rent is specified in each property card of the land. As the user advances in the game the rents become higher. The owning user has the choice to build houses in the

properties that they own in order to increase the rent. After building four houses, the user can upgrade to a hotel, which has a considerably higher rent.

- There are special squares in the board:
 - Tax Squares: the user has to pay some money to the bank.
 - Utility Squares: the amount of rent is 40 times the sum of the dice roll if only one is owned and 100 times the sum of the dice if both are owned.
 - Railroads: the rent is 250 if one is owned, 500 if two, 750 if three and 1000 if all four are owned.
 - Chance and Community Chest: when landing in these squares the user picks a chance or community chest card which can have instructions like pay a tax, gather some money, Get-Out-Of-Jail cards, move to a certain square etc.
- Every time the users pass through the starting point they collect some money.

2.2. Functional Requirements

Help

The players should be able to view Help before starting playing a game. This panel will display the official rules of the game with the necessary additions or modifications that we are providing in this smaller version of μ Monopoly. It will describe all the possible scenarios of how the game might turn out and what choices will the players have to progress in the game for those particular conditions. On the other hand, the players should be able to access Help while playing the game as well without interrupting it.

Starting game

As soon as the users will initiate the software, they will see the option of starting the game. This is going to initiate the basic functions for the game and make the environment ready for starting the interactions.

Upon the game starting, each player will receive 1500\$ to start off, however every time they pass from the GO square they receive 200\$ as salary.

Profile selection

The users can manage their profile by assigning a name to the profile and choosing a token. Tokens will come in a variety of shapes: Scottish terrier, Battleship, Automobile, Top Hat, Thimble, Shoe, Wheelbarrow, and Iron, with the explanation of each token in order to assist their decision making. The tokens are upgradable for extra bonuses during the game and they should express the personality of the player.

Playing the turn

The players will be able to advance in the game by rolling the dice and moving according to the sum received.

After moving the token to the destination, the players will eventually land in a property square or special square. If it is a property square, the players can buy it if it is not taken, sell it if they own it

and they are running out of money, build a house if they own all the properties of the same color, build a hotel if they already have four houses or pay the rent if it already has an owner.

If all the properties of the same color are owned by some other player, the player who lands in this property will have to pay twice the assigned amount, or if it has houses / hotels, they need to pay for every of those with the sum indicated in the property cards.

If the players land in a tax square, they need to pay the indicated amount to the bank.

If the players land in a Utility Square they can buy the property but they cannot build neither houses nor hotels. This special square makes the landing opponent pay 4 times the dice roll and if both Utilities are owned, the opponent will pay 10 times the dice roll.

If the players land in the Railroads squares, they can purchase it as well but they cannot build houses or hotels. If the player owns only one Railroads property, the opponent will have to pay 25\$ upon landing there and 50\$ in case they own 2, 100\$ in case they own 3 and finally 200\$ in case they own all four of them.

If the players land in GO square or just pass over it they receive 200\$ as salary.

If the players land in Jail / Just Visiting square, they have nothing to do: the property cannot be owned.

If the players land in Free Parking square, they have nothing to do: the property cannot be owned.

If the players land in Go To Jail square they go to Jail and stay there for 3 turns, pay the Bailout or use a Get-Out-Of-Jail card to get out.

Upgrading the dice

The players will be able to purchase a dice upgrade: golden or platinum. Both of these dice will provide facilities or bonuses to the player and they can be purchased by paying a relatively large amount of money.

The golden dice will allow the player to roll twice and the platinum dice will allow the player to move twice the sum of the dice roll.

Upgrading the token

The players should also be able to upgrade the token by purchasing it with a relatively large amount of money.

Gold tokens decrease the amount of rent which is needed to pay to the competitor by 10% and the platinum token decreases the amount of money the user needs to pay in order to purchase the land, build houses and build hotels by 10%.

Buying or selling the property

In order to buy the property, the player who lands in that square should have enough money to purchase it with the sum indicated in the property card. If they decide to buy it, the price will be deduced from their account and it will go to the bank (which is abstract but has infinite amount

of money). If the player wants to progress to building houses or hotels, they need to own all properties of the same color.

If the player wants to sell the property, all the houses and hotels will be destroyed and they will give the property back to the bank and receive half of the amount needed to buy it.

Building houses or hotels

In order to build a house the player needs to own all the properties of the same color. After paying the necessary amount of money to the bank, they can purchase a house. In order to build a second house on a certain property, all the other properties of the same color should have one house each and so on.

There is a maximum of 4 houses that can be built in a property, afterwards, if the player wants to build more they can only sacrifice all four houses and build a hotel.

Quitting game

The players will be able to quit the game at any time: during the game or upon finishing. If one of the players leaves the game, the game will not be saved. If the game is finished, the players will be automatically redirected to the Home screen when they can start a new game.

Saving high scores

The application will save the score of the winner player attached to the winning player's profile (name and token choice). If the game has not finished and the players just quit, they won't be able to save their scores.

2.3. Non-functional Requirements

2.3.1. Usability requirements

- The players must easily adopt game rules by providing clear rules panel with examples as well.
- The software must be user-friendly:
 - o Easy to use: the players will be able to easily find a button when they are looking for it.
 - o Assistance: the players will be presented with familiar colors such as green buttons for Yes and red ones for No options.
 - o Monopoly branding: the players should have the same Monopoly board image with the same logo, colors, naming and rules as the original game.
- The players will be limited from 2 – 4 as an enforcement to the original game rules.

2.3.2. Performance requirements

- The software must never freeze and interrupt the game.
- The software must be able to handle the amount of throughput necessary to make a smooth continuation of the game.
- The memory occupied during the run time should be minimized with no heavy components besides the interface drawing library.

2.3.3. Supportability requirements

- Adaptability requirement: The software must come as a standalone executable file which does not require installation.
- The software should be backwards compatible with the older versions.
- The software should represent the US Standard Edition of the Monopoly game.
- Portability requirement: The software will be able to run on every platform and executable in the same manner. Its underlying core should be a Java Archive file.

2.4. Constraints (Pseudo-requirements)

2.4.1. Implementation requirements

- The programming language will be Java and the graphics will be created using Adobe Photoshop CC software.

2.4.2. Packaging requirements

- The software should come as an .exe file with our custom icon which also represents the Monopoly logo. The file should be a JAR wrapper.

2.4.3. Legal requirements

- The software should come as an open source software licensed under MIT License for free redistribution.

2.5. Scenarios

S1

Scenario name: Starting the game

Participating Actors: buse:Player

Flow of events:

1. Buse wants to start the game, so she makes sure at least one other player is willing to play, but she can't play with more than 3 other players. Afterwards she starts the application by double clicking on the file. She wants to make sure that she knows all the rules, so she opens the Help panel.
2. In the Help panel she finds that there is a list of all the rules illustrated with pictures. She scrolls through all the rules and reads them.
3. She then moves on to starting the game. She goes to the Home Page and she clicks the Start Game button. As soon as she is introduced to the new panel, she registers her name

and chooses a token. Then, all the other players do the same thing and thus create their profile. The system gives every player 2000\$ as a budget to start off.

Let the game begin!

S2

Scenario name: Play Turn

Participating Actors: buse:Player alper:Player

Flow of events:

4. Buse presses the button to roll the dice. The result of dice are 4 - 5, totally 9.
5. Buse moves the token for 9 squares.
6. The square is a town named Keçiören. Buse may buy this place but she decided not to.
7. Alper rolls the dice. The result of dice are 6-6, total 12.
8. Alper's token move on 12 square. The landing square is a Railroad. Alper wants to buy this Railroad and he presses on the 'Buy' button. Alper is able buy this place because no other player owns it. In the screen, the cost of the Railroad is displayed as 200\$ and this amount is deduced from Alper's budget making him the owner now.
9. Alper has the privilege to roll the dice again because he rolled doubles previously. The new result of dice is 1 - 1.
10. Alper's token moves on 2 squares further to a Chance card square. Random Chance card is seen in the screen. As the card describes, Alper wins 1.000\$ extra and it makes him happy.
11. Alper rolls the dice again, in this time, the result is 2 - 2. Because of the rule stating that rolling three doubles sends the player to jail, Alper's token goes to jail and his turn ends.
12. It is the Buse's turn and she rolls the dice with the result 2-1.
13. Her token moves on 3 squares to the square which is already owned by Alper. Buse has to pay the cost of rent to Alper.
14. Thus her turn ends and it is back to Alper to try and roll doubles in order to get out of jail as he doesn't have a Get-Out-Of-Jail card and he refuses to pay the bail.

S3

Scenario name: Upgrade dice or token

Participating Actors: buse:Player alper:Player

Flow of events:

1. Buse wants to play μ Monopoly, so she clicks the Start Game button. The system asks Buse how many players will participate in the game, and Buse decides to play the game with Alper. Since the number of players is not less than 2, they can provide the first condition of starting the game.
2. Then system asks Buse and Alper for creating their own profile by entering their names and choosing their tokens to play.
3. Upon starting to play the game, Buse prefers to upgrade her dices and Alper his tokens in order to increase the chance of winning. When Buse clicks on the Upgrade Dice button, a new panel comes up and ask her the type of upgrade (golden or platinum). If she has enough money the purchase is complete and the dice type is changed to golden for her.
4. In the next turn Buse rolls the purchased type of dice. The feature of golden dice is rolling with multiplicity of 2. For the platinum token, the multiplicity becomes 3.
5. For upgrading the token, Alper can choose golden or platinum token and follows a very similar procedure. The golden token has 10% off for buying property and platinum token has 10% off for building.
6. As Buse rolls the golden dice that she bought, she gets 3 and 4 and therefore she moves 14 squares as it is the special ability of the upgraded dice.
7. Afterwards Alper rolls and he obtains 2 and 3, but he moves 5 as his dice is ordinary. He lands on a square that he didn't own before and which normally costs 200\$ but he buys it for 180\$ as he owns a golden token. He ends his turn and the game goes on.

2.6. Use Case Models

Use Case diagram

The use case diagram is shown in Figure 1 and the following are the textual case descriptions:

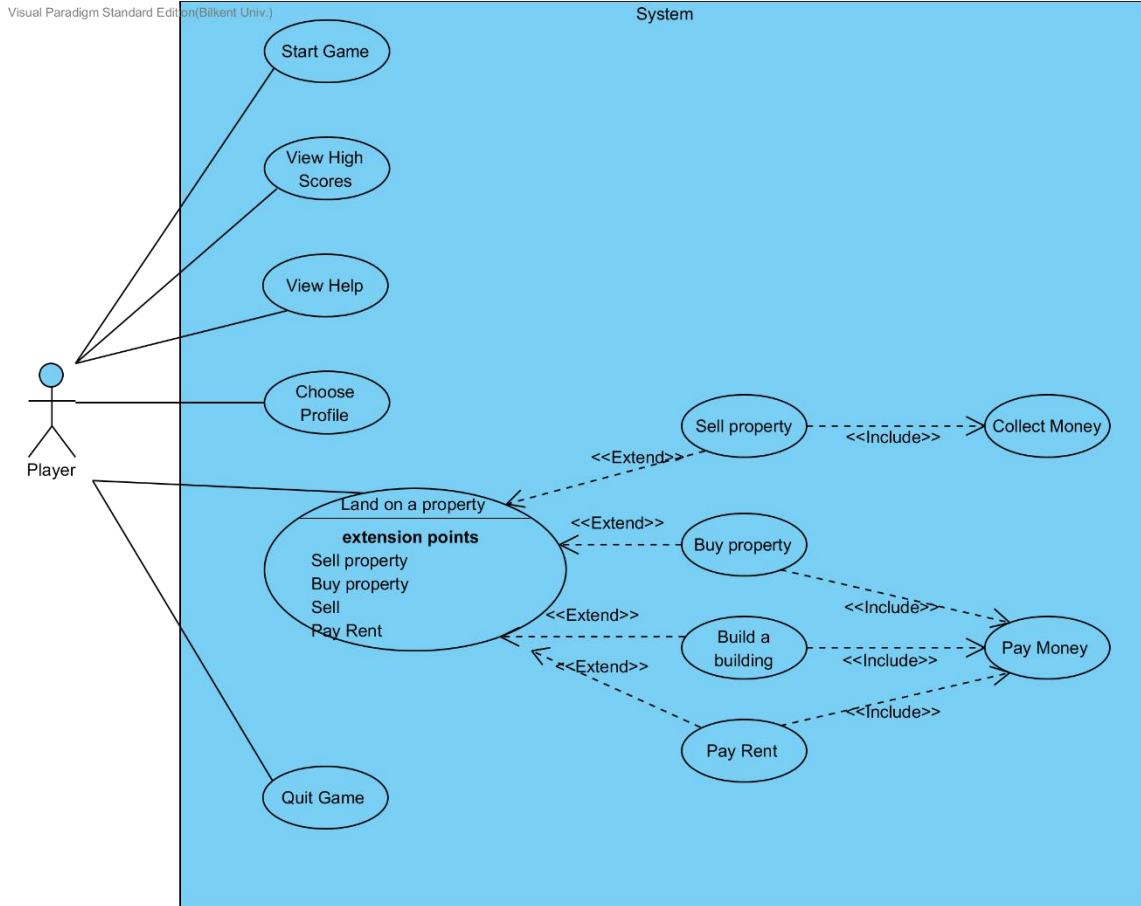


Figure 1 - Use Case Diagram for μ Monopoly

Case #1

Name: Start Game

Participating Actors: Initiated by: Player.

Flow of events:

1. As soon as the Player starts the application of μ Monopoly and the system introduces him with the Home Screen
2. The Player selects the Start Game button in order to initiate the procedures of creating the game.
3. The system allows every Player to enter their name and choose their personal token in turns.
4. The system makes sure no two players choose the same token and name.
5. After all the Players have registered in the game by creating their profile, the system initiates the game.
6. The system gives every Player 2000\$ as a starting budget.
7. The system prompts every Player to roll the dice and assigns the Player who rolled the biggest sum to start first.
8. The system keeps then track of the turns in the order of registration.

Entry Condition: The Player has opened the application

Exit Condition: Each Player has created their profile and they are ready to play the game in a determined order.

Quality Requirements: Creating a Player profile (name and token) should not take more than 1 second.

Case #2

Name: View High Scores

Participating Actors: *Initiated by:* Player.

Flow of events:

1. When the player opens the application, they see the main screen (Home Screen).
2. In main screen, there are some selection buttons as: Start Game, High Scores, Help and Quit.
3. They click the High Scores button to see ranked scores of played games.
4. Score table is seen like, name of the winner, their token, final money and the date of game.

Entry Condition: The player has opened the application.

Exit Condition: The player has to click the back button in bottom left of the screen.

Quality Requirements: The system should be able to retrieve the Top 8 results in less than 1 second.

Case #3

Name: View Help

Participating Actors: *Initiated by:* Player.

Flow of events:

1. When the player opens the application, they see the main screen (Home Screen).
2. In main screen, there are some selection buttons as: Start Game, High Scores, Help and Quit.
3. Player clicks the Help button to see help pages which include information of the game.
4. In the Help pages, the player can see how to play game, the rules, and any additional information about game.
5. The player can scroll the page from the scroll bar in the right side of the panel.

Entry Condition: The player has opened the application.

Exit Condition: The player has to click the back button in bottom left of the screen

Case #4

Name: Choose Profile

Participating Actors: *Initiated by:* Player.

Flow of events:

1. When the players select the Start Game button, the system allows players to enter their names
2. The system allows players to choose their tokens in order to use in turns of μ Monopoly.
3. The system controls whether no players have the same token and names in order to start the game.

Entry Condition: The players have started the application have selected the Start Game button.

Exit Condition: The players can now start playing the game in turns.

Case #5

Name: Land on property

Participating Actors: *Initiated by:* Player.

Flow of events:

1. As soon as the players start the application of μ Monopoly they roll the dice and move the token the required number of squares.
2. When the players land on a property, the system checks whether they own the property.
3. If they do not own the property, the system asks them whether they want to purchase it.
4. If the players want to buy the property, they pay money to the system.
5. If they do not buy the property, their turn ends the game continues.
6. When the player lands on a square which is already theirs, the system allows the players to build on it or sell it.
7. If he decides to sell the property, the system gives the money back to the player.
8. If the player lands on a square that is owned by another player, they pay rent of the property to the owner.
9. If the players want to increase the rent of their property, the system allows them to build houses or hotels on their own properties.

Entry Condition: The players have opened the application.

Exit Condition: The player has ended their turn.

Case #6

Name: Quit Game

Participating Actors: *Initiated by:* Player

Flow of events:

1. The system checks if the game has finished or not.
2. In case the game has finished, the winner's profile will be saved in the High Scores section accompanied by the date, time and the time length of the game. Otherwise the system will only generate a warning dialog asking the Player whether they really want to quit the game as it has not finished yet.
3. The system logs out and discards any other information such as the profile of the players that did not win.

Entry Condition: The Plyer presses on Quit Game button.

Exit Condition: The system is off

Quality Requirements: Saving the High Scores should not take more than 1 second. The system should hide the window before terminating the process.

2.7. User Interface

The following will be a series of screen mockups for the μ Monopoly software:

Home Screen

The screen mockup shown on Figure 2 is going to be the first screen to be presented to the users as soon as they start the application. It shows the logo of the software at the top, four buttons for user's choices and the credits at the bottom of the screen.

The four buttons give the user the ability to:

1. Start Game: This is going to redirect them to the Player Registration Screen and then they will be able to continue to the game.
2. High Scores: This is going to redirect them to the High Scores Screen which displays the top 8 players' rank, name, token, amount of money for winning and lastly their time of winning.
3. Help: This is going to redirect them to the Help Screen where they will be able to see the rules of the game.
4. Quit: This is going to close the software and exit the user.



Figure 2 - Home Screen mockup for μMonopoly

Player Registration Screen

Figure 3 represents the Player Registration screen which is accessed from the Home Screen and leads further to the Main Screen where the players can play the game.

This screen starts with 2 dividers for entering the players' names and selecting the token. In case there are more than two players, the "+ Add more players" button will add another space upon clicking. However there is a limit of four players for this game, therefore after four dividers have been displayed this button will disappear.

After every player has entered their names and clicked on a token picture to select their favorite one, they can progress by clicking the right arrow button at the bottom right corner, which will display the Main Screen.

Main Screen

Figure 4 displays the Main Screen. It contains two main dividers: the left, Monopoly board and the right, Sidebar.

The Monopoly board divider contains all 40 squares for the Monopoly game with their names, optionally pictures and town colors. In the squares there are going to be four places for the token picture which displays the current players' positions. The houses / hotels built are going to be

Register Players

Please select the players' names and tokens:

Player #1:

Name: John



Player #2:

Name: Kate



Player #3:

Name: Ben



+



Figure 3 - Player Registration Screen mockup for μ Monopoly

FREE PARKING

KENTUCKY AVENUE

?

INDIANA AVENUE

ILLINOIS AVENUE

ATLANTIC AVENUE

VENTNOR AVENUE

WATER WORKS

MARVIN GARDENS

GO TO JAIL

Roll Dice

End Turn

Buy

Sell

Build

3

VENTNOR AVENUE

RENT 22\$

With 1 House

110\$

With 2 Houses

330\$

With 3 Houses

800\$

With 4 Houses

975\$

With HOTEL

1150\$

Mortgage Value 130\$

Houses cost 150\$ each

Hotels 150\$ each

plus 4 houses

If a player owns all the sites of any color group, the rent is doubled on unimproved sites in that group.

JAIL

CONNECTICUT AVENUE

VERMONT AVENUE

?

ORIENTAL AVENUE

READING RAILROAD

INCOME TAX (200\$)

BALTIMORE AVENUE

MEDITERRANEAN AVENUE

GO

PLAYERS

1. John (Dog)

1200 \$

2. Ben (Shoe)

500\$

3. Leslie (Hat)

480\$

4. Kate (Thimble)

120\$

Upgrade Token (2000\$)

Upgrade Dice (3000\$)

MEDITERRANEAN AVENUE

BALTIMORE AVENUE

ORIENTAL AVENUE

VERMONT AVENUE

CONNECTICUT AVENUE

ST CHARLES PLACE

STATES AVENUE

VIRGINIA AVENUE

ST JAMES PLACE

TENNESSEE AVENUE

NEW YORK AVENUE

KENTUCKY AVENUE

INDIANA AVENUE

ILLINOIS AVENUE

ATLANTIC AVENUE

VENTNOR AVENUE

MARVIN GARDENS

PACIFIC AVENUE

CAROLINA AVENUE

PENNSYLVANIA AVENUE

PARK PLACE

BOARDWALK

ELECTRIC COMPANY

WATER WORKS

READING RAILROAD

PENNSYLVANIA RAILROAD

B&O RAILROAD

SHORT LINE

View Help

Quit Game

Figure 4 - Main Screen mockup for μ Monopoly

displayed in the color ribbon of the town squares, therefore there is going to be a chain of four places for the buildings too in that part of the square.

In the middle of the Monopoly board there are going to be the following sections:

- Roll Dice, End Turn buttons and Dice Display divider at the top.
- Buy, Sell and Build buttons further bottom
- Property Card of the current player's current position and if it is a town, the number of houses built or an indication that a hotel was built. In case the current player is in a Chance or Community Chest square this will be substituted by the randomly selected card. For the other squares this is going to be invisible.

On the Sidebar the following elements will be displayed:

- List of players and their current budget. The player in the current turn will be highlighted.
- Two buttons for upgrading the dice and the token showing the necessary amount of money needed for the purchase.
- The grid of all possible property cards is further down. The ones that are owned by the current player will be in full color, while the others will be in grayscale.
- The button for accessing the View Help panel which will redirect back to this.
- The button for quitting the game which will issue a dialog for confirmation before ending the game.

Help Screen

Figure 5 is a mockup for the Help Screen which should provide the users with the rules of the games as described in the US Standard Edition Monopoly game. The main panel will be scrollable and there will be a back button which will send the user back to the previous screen, either the Home Screen or Main Screen, depending on how the player accessed it.

High Scores Screen

Figure 6 is a mockup for the High Scores screen which shows the top 8 players of all the time as long as the software has been used in the same machine.

It displays the player's rank, chosen name, chosen token, the amount of money they had when they won and the date of the game.

In addition there is a button allowing the user to go back to Home Screen as this can be the only origin of coming into this screen.

Help

Rules

1. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed vitae pretium turpis. Duis vitae libero ut dui venenatis cursus. Aenean in orci tincidunt, scelerisque risus tristique, tincidunt magna.
2. Pellentesque vestibulum mauris vitae lorem rutrum egestas. Morbi pulvinar, orci id tincidunt tincidunt, nulla eros interdum felis, in molestie nisi lacus eu sapien. Cras eget vehicula lacus. Vivamus sed elit eleifend, tristique sapien eu, vehicula augue.
3. Sed efficitur, quam eget blandit varius, velit magna vestibulum ex, a commodo lorem orci sed tellus. In hac habitasse platea dictumst. Curabitur mattis ut augue id interdum. Integer eu vehicula lacus. Praesent vitae risus erat. Etiam eget consectetur tortor, a vehicula felis.
4. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec eget posuere mauris, ac sollicitudin lorem. Aliquam tempor felis quis lectus bibendum, vulputate vestibulum justo aliquam. Quisque lacinia auctor interdum.
5. Mauris varius nibh neque, vitae tristique mauris hendrerit mattis. Integer ut mattis elit, id facilisis augue. Etiam tincidunt nisi sit amet ipsum tincidunt, ullamcorper aliquam lacus tristique.



Figure 5 - Help Screen mockup for μ Monopoly

High Scores









1. John		5000\$	21/08/2015
2. Ben		3200\$	02/09/2015
3. Sam		2800\$	25/11/2015
4. Kate		1200\$	25/10/2015
5. Nick		1100\$	21/08/2015
6. Lily		1000\$	02/09/2015
7. Sam		800\$	25/11/2015
8. Amy		780\$	25/10/2015



Figure 6 - High Scores Screen mockup for μ Monopoly

3. Analysis

3.1. Object Model

3.1.1. Domain Lexicon

BonusCard: Bonus cards are the two types of cards drawn in Monopoly. Bonus cards consist of two types: Chance cards and Community Chest cards.

Dice: Dice are the main form of movers in Monopoly. They are the standard, 6-sided white cubes with black pips. Some editions will come with specially dice types (golden and platinum).

DiceType: There are two types of dice: golden and platinum. They differ in the way they offer bonuses about the count of squares to move.

ChanceCard: This is one of two types of bonus cards. Chance cards are orange and are placed near the GO Square. A Chance card is more likely to move players, and may often come with lethal consequences. Chance cards include "Go back 3 spaces", "Go to jail" and "Get out of jail free" cards.

CommunityChestCard: This is the other kind of bonus cards. Community Chest cards are most likely to give you money. They are blue and sit next to Free Parking square. Community Chest cards include "Go to jail" card and "Get out of jail free" card.

Player: Player class holds name, token, budget, dice type, token type, position of the player and control that player is in jail or not. It is a class of the main and only actor in the game.

PropertyCard: In the game of Monopoly, the winning objective is to make all the opponents go bankrupt. Properties are necessary to achieve this goal. Property cards include TownCard, UtilityCard and RailRoadsCard. Properties may be bought in one of 3 ways: by landing on the property space and buying it. When a player buys or otherwise gains possession of a property, he or she receives the property's corresponding property card, which lists all relevant information on the property. When a player owns all the properties in a color group, she or he is said to have a monopoly, which allows the player to charge double rent or build it up with Houses and Hotels. Property costs and rents escalate as the player rounds the board. Properties range anywhere from \$60 to \$400, while rents can range from \$2 to \$2000.

PropertyCard: Any card that proves the ownership of a player's square and contains the information about the rent to be taken from the opponent upon landing.

RailRoadsCard: This is a special type of property card which is particular to RailRoads squares. It contains information about the rent with one, two, three and four owned stations.

TownCard: This is a special type of property card which is particular about town squares. It contains information about the rent of the property with no house or one, two, three or four houses or with a hotel. It also contains the price of building a house or hotel.

UtilityCard: This is a special type of property card which is particular about utility squares. It contains the information about rent extraction from the player and the multiplication factor of the roll.

MonopolyBoard: Monopoly Board include 40 squares. In addition this has the dice needed for the player's advancement in the game.

Square: Represents any 40 place of the monopoly board. Squares consist of properties, free squares, jail squares, tax squares and card squares.

ChanceCardSquare: A square requiring to pick a ChanceCard.

CommunityChestCardSquare: A square requiring to pick a CommunityChestCard.

CardSquare: A square requiring to pick a BonusCard and it is the parent class of CommunityChestCardSquare and ChanceCardSquare.

PropertySquare: A square that can be owned and therefore be profitable for rent extraction from the opponents.

RailRoadsSquare: It is a derivative of the PropertySquare class. These kinds of properties cannot have buildings but their special feature is the fact that more of these RailRoads that a player owns the more the opponent has to pay to them. (E.g. the landing player has to pay 50\$ if the owner only has that particular square, 100\$ if two, 200\$ if three and so on).

TownSquare: It is a derivative of the PropertySquare class. These kinds of properties can have buildings (houses and hotels) and are distinguished by a color.

UtilitySquare: It is a derivative of the PropertySquare class. These kinds of properties cannot have buildings but their special feature is the fact that when a player lands there they pay a certain multiple of the dice roll to the owner. If two of these kind of properties are owned then the multiplication factor is larger.

Token: Tokens are the playing pieces used in the Monopoly game.

TokenType: There are two types of token: golden and platinum. TokenType represents the special ability related to the token such as discount for rents and buildings.

3.1.2. Class Diagram

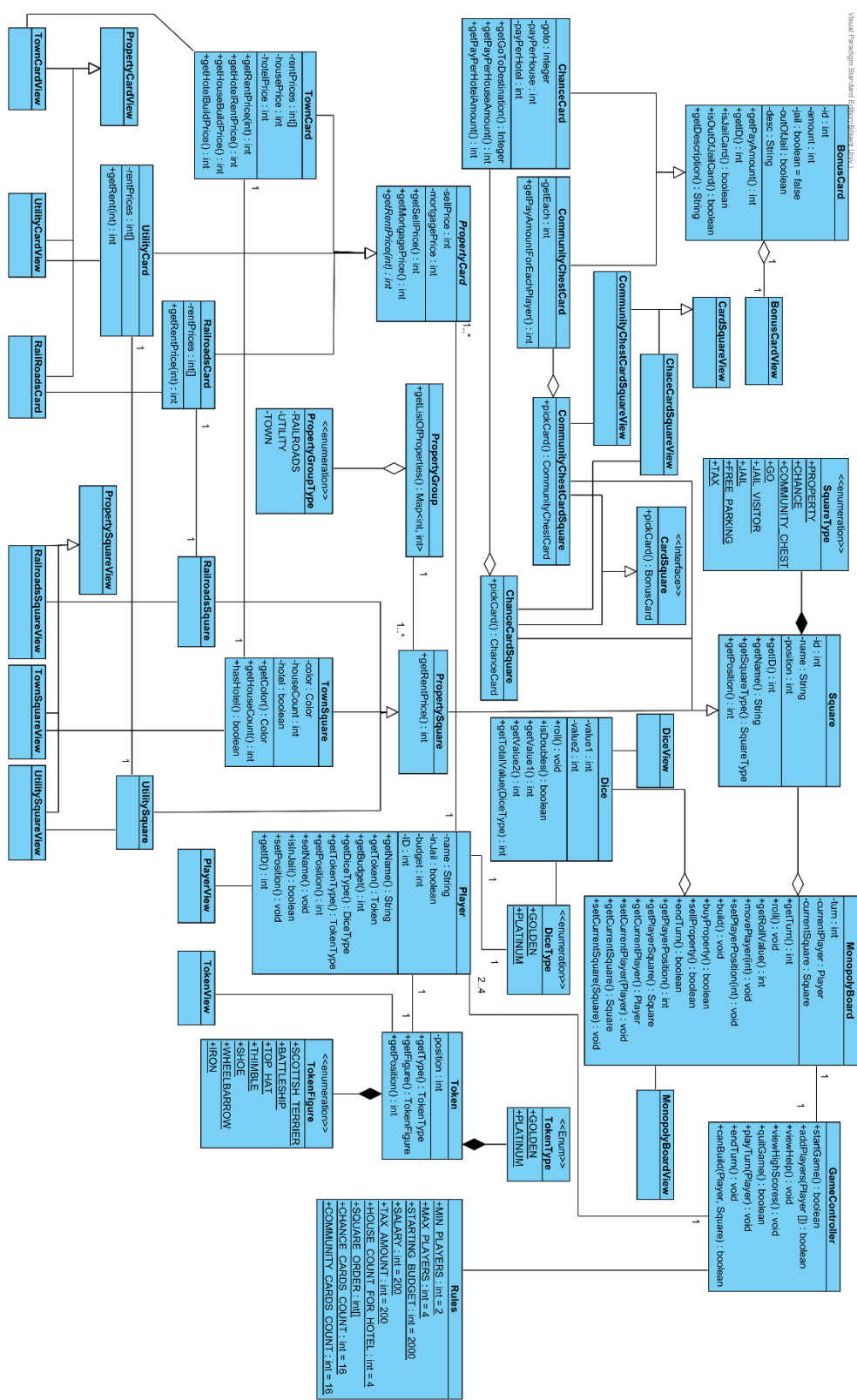


Figure 7 - Class Diagram for μ Monopoly

The above is a class diagram for the μ Monopoly software. The main components of this diagram are as follows:

- The tiles of the Monopoly board are all called Squares and this acts as a parent class for all. The children of the Square class are ChanceCardSquare, CommunityChestCardSquare and PropertySquare. These are the only types of squares that show a different and interesting behavior from the plain Square class. They correspond to the respective types of Squares in the Monopoly board, and PropertySquare is the kind of Square that can be owned. This includes TownSquare, UtilitySquare and RailRoadSquare. The difference between those is that a TownSquare is a PropertySquare where the owner can build houses and hotels. The Utility squares define a different kind of rent with respect to the dice and RailRoadsSquare bases the rent on the count of owned RailRoads.
- In addition the other classes included in the diagram are the ones intuitively considered such as: dice, cards, token and player. The top class 'encapsulating' the squares, cards, property cards and dice is the MonopolyBoard class. The top level GameController is the one which only has a reference to the MonopolyBoard class and the Player instances. It is the higher level coordinator which knows the turn, responds to the stimuli obtained from the user through the views and coordinates the Monopoly Board elements which are mutated by forwarding methods.

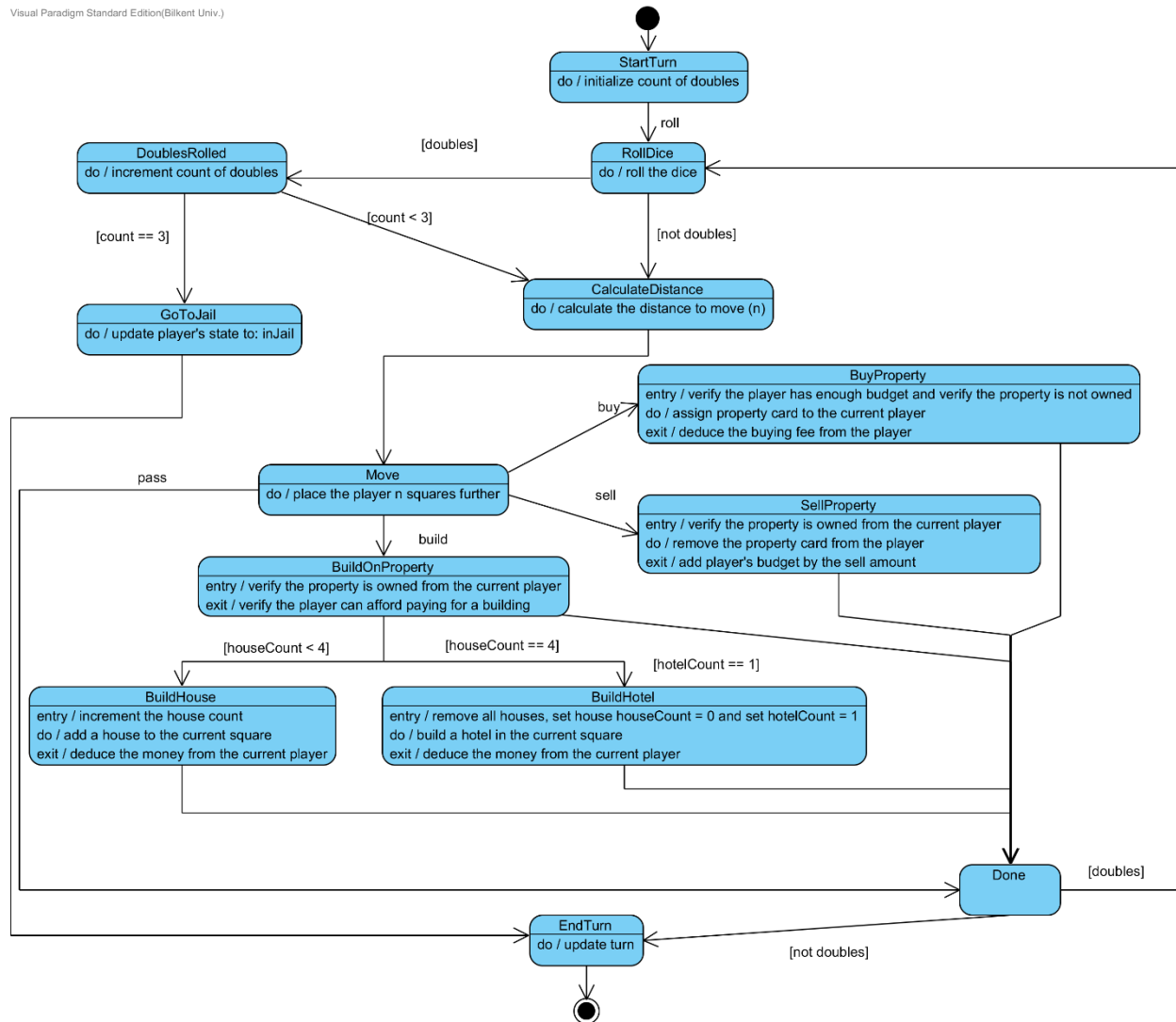
3.2. Dynamic Models

3.2.1. State Chart

The following is the state chart diagram of the behavior of MonopolyBoard class which is the principal component of player interactions with the game.

The diagram in Figure 8 describes the typical behavior of the MonopolyBoard class at a certain player's turn. It describes the series of the events that trigger certain changes in the state of the class and generate actions.

As soon as the player starts the turn, the count of doubles is initialized to zero and on request to roll (a response from a button click) the MonopolyBoard initiates a dice roll which selects two random values from the range 1 to 6. In case the player rolls double, they have the ability to roll again after moving to the certain square according to the dice value. When there, the player can choose to buy, sell or build. If there are already 4 houses built, they can upgrade to hotel, otherwise they can only build a house. During all these states the class should verify the owner of the property, their budget and if the property can have houses or hotels (unlike RailRoads or Utility Squares). The class also check whether the player rolls three times doubles, in which case they are sent to Jail. At the end of the turn the player's turn ends, thus the turn is updated to point to next player.

Figure 8 - State Chart Diagram of MonopolyBoard class for μ Monopoly

3.2.2. Sequence Diagrams

Diagram 1

The diagram in Figure 9 describes the sequence of operations for the starting of a new game. It describes the initialization of the Player and MonopolyBoard objects from GameController class. Afterwards, the MonopolyBoard object initializes all 40 Square objects and aligns them in the order given by their position attribute. In addition the Dice object is also initialized and a new turn is initiated by the startTurn() function call to the MonopolyBoard.

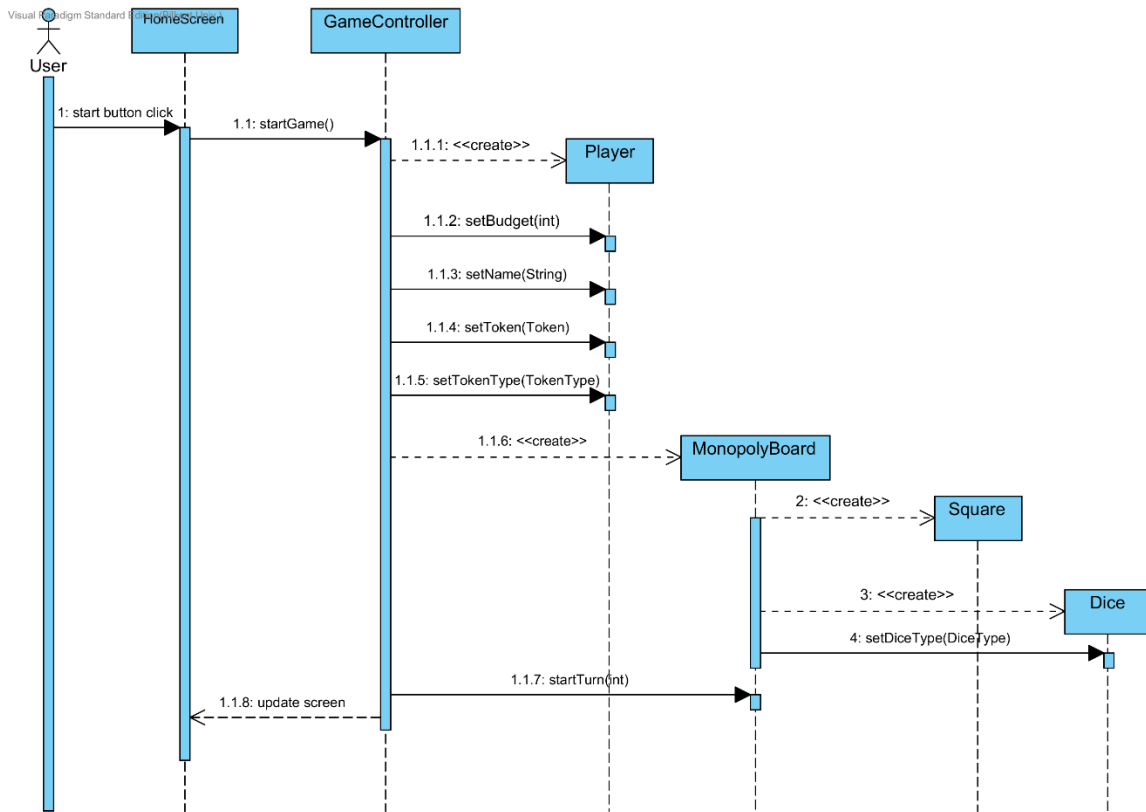


Figure 9 - Sequence Diagram for starting the game in μ Monopoly

Diagram 2

The diagram in Figure 10 expressed the sequence of operations needed to upgrade the player's token. Upon the Player's stimulus to upgrade the token, the GameController initiates the operations and assigns the new tokenType attribute to the Player object provided the money needed for the upgrade is enough and it has been deduced from the player's budget.

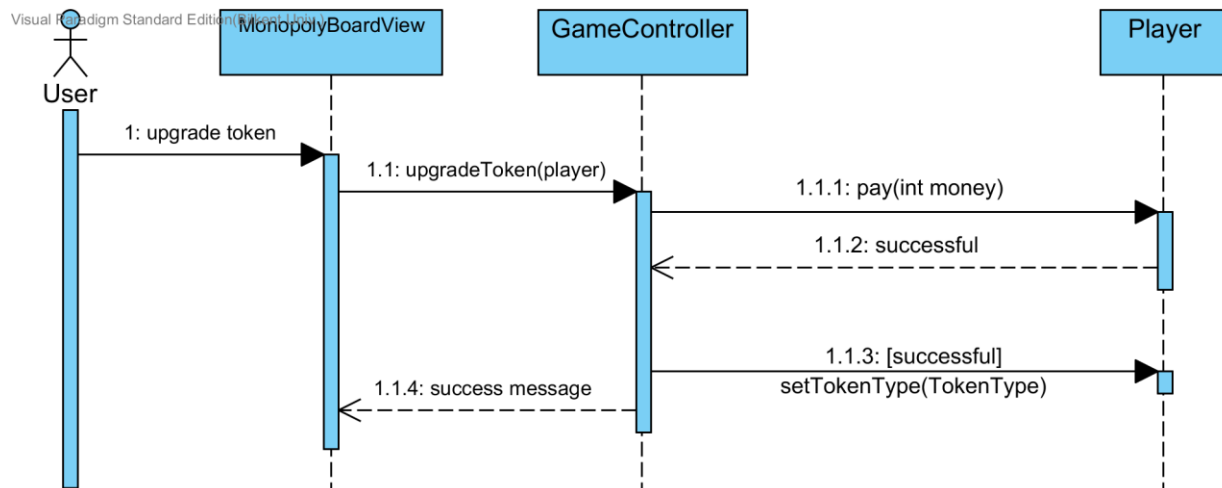


Figure 10 - Sequence Diagram for upgrading the token in μ Monopoly

Diagram 3

The diagram in Figure 11 expressed the sequence of operations needed to perform a turn play. In an abstract, high-level approach it describes the typical operation call from MonopolyBoard class to the assisting classes such as Dice and Player, which owns a token and the latter acts as a reference to the current location of the player.

The MonopolyBoard class reacts to the call from GameController which initiates a new turn and makes the dice roll two random values between 1 and 6. Afterwards it calls the Dice class's `getTotalValue()` method to get the amount of squares they need to move and makes the player move this many squares eventually.

In case doubles were rolled, the rolled value was doubles, the Player has the chance to roll again and the procedure starts over.

NOTE: The behavior of the MonopolyBoard class is also described by a State Chart diagram in section 3.2.1 and it provides a deeper insight to the sequence of operations and the activities depending on the output.

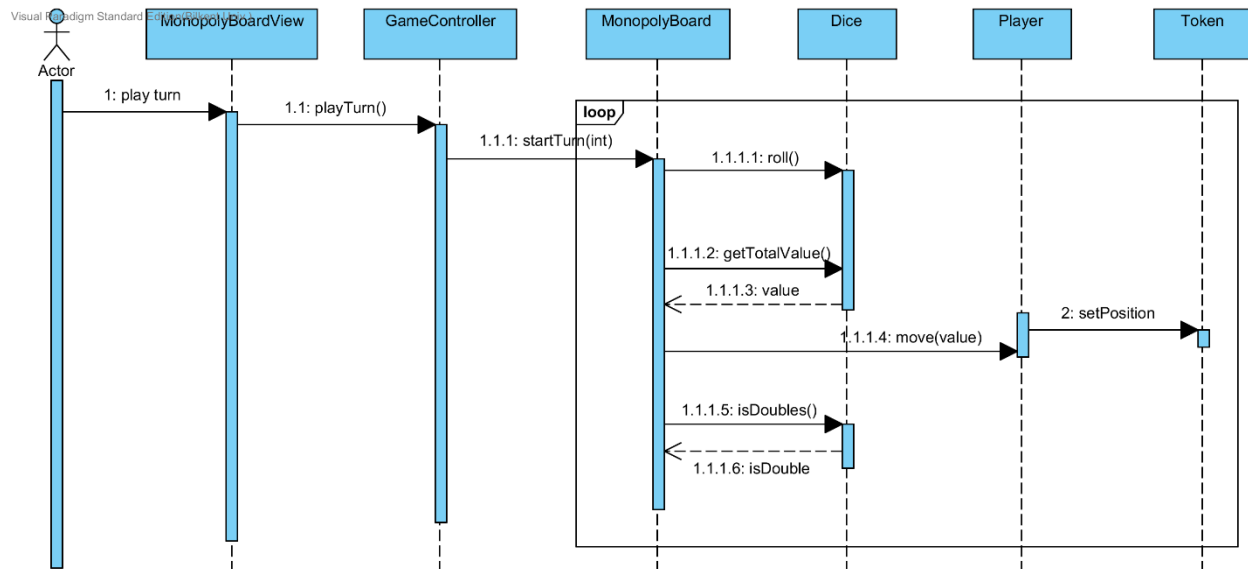


Figure 11 - Sequence Diagram for playing a turn in μ Monopoly

Diagram 4

The diagram in Figure 12 expressed the sequence of operations needed to upgrade the player's dice. Upon the Player's stimulus to upgrade the dice, the GameController initiates the operations and assigns the new diceType attribute to the Player object provided the money needed for the upgrade is enough and it has been deduced from the player's budget.

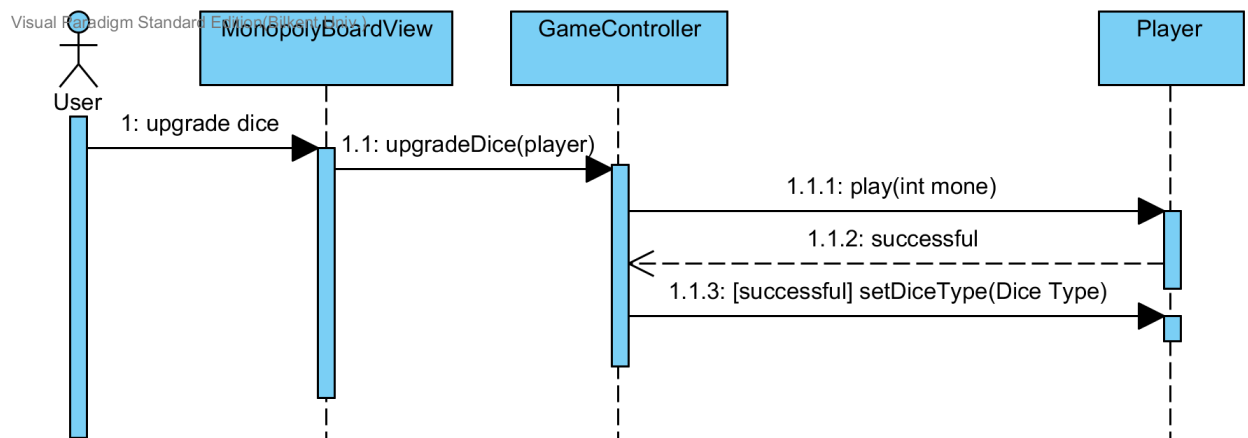


Figure 12 - Sequence Diagram for upgrading the dice in μ Monopoly

4. Design

4.1. Design Goals

PERFORMANCE CRITERIA

Response Time

μ Monopoly game is powered by Java programming language. In μ Monopoly game, we will work on minimizing the response time and avoiding the setbacks that come with the delays due to the Java language constructs

DEPENDABILITY CRITERIA

Reliability

In order to have reliable game, our main aim is minimizing the number of errors. In order to achieve this, μ Monopoly will be implemented using object-orientation principles such as encapsulation, which contributes in the error minimization due to the privilege limitations.

MAINTENANCE CRITERIA

Extensibility

Extensibility is an important aspect for our project and it is related to our MVC design as for instance adding more attributes to a model or attaching more views to it will be a trivial process.

Adaptability

μ Monopoly game will be developed in Java, therefore granting the ability to run in the same manner in different environments, systems and conditions.

Portability

Portability is about deploying the software in different environments with the aid of good packaging such as Java Archives (JAR) files. It should eventually result in a standalone executable file.

Readability

During the implementation we aim to provide very readable code that will be easily understood from every member of the developers' group and thus easily modifiable.

Traceability of Requirements

While developing we have to make sure that every requirement is traceable back to the features provided by the software.

END-USER CRITERIA

Usability

Usability focuses on user rather than the system. μ Monopoly game should be easy to learn and to use by following the common user experience patterns from similar games / software.

4.2. Sub-System Decomposition

In order to organize all the classes in our project in a way that they are grouped in entities that represent a uniquely working model, we proceed to the sub-system decomposition. We designed the structure of the system in a way that classes that represent similar functions and use similar contexts belong in one system component. In addition, we worked on minimizing the dependencies of such components to ensure maintainability. The above principals that we follow have led to the following system as depicted in the Figure 13 as a reflection of lowly coupled systems with high cohesion.

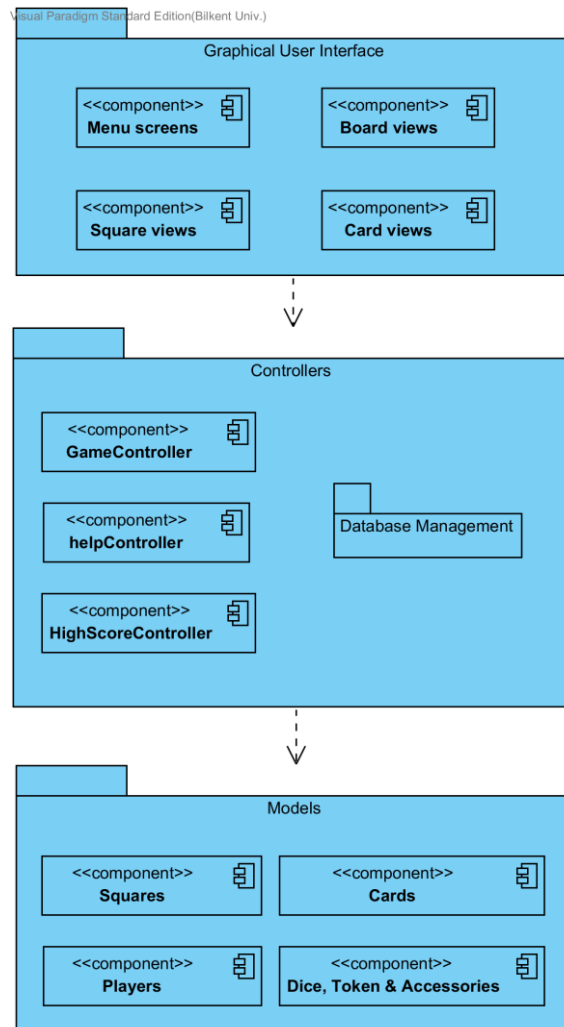


Figure 13 – Component diagram for sub-system decomposition

There are three main systems: GUI, Controllers and Models which group the views, models and controllers elements in three layers respectively according to the MVC architectural principle. The following will be a detailed look at each sub-system.

4.2.1. Graphical User Interface Sub-system

The Graphical User Interface sub-system will contain all the views of the software as follows:

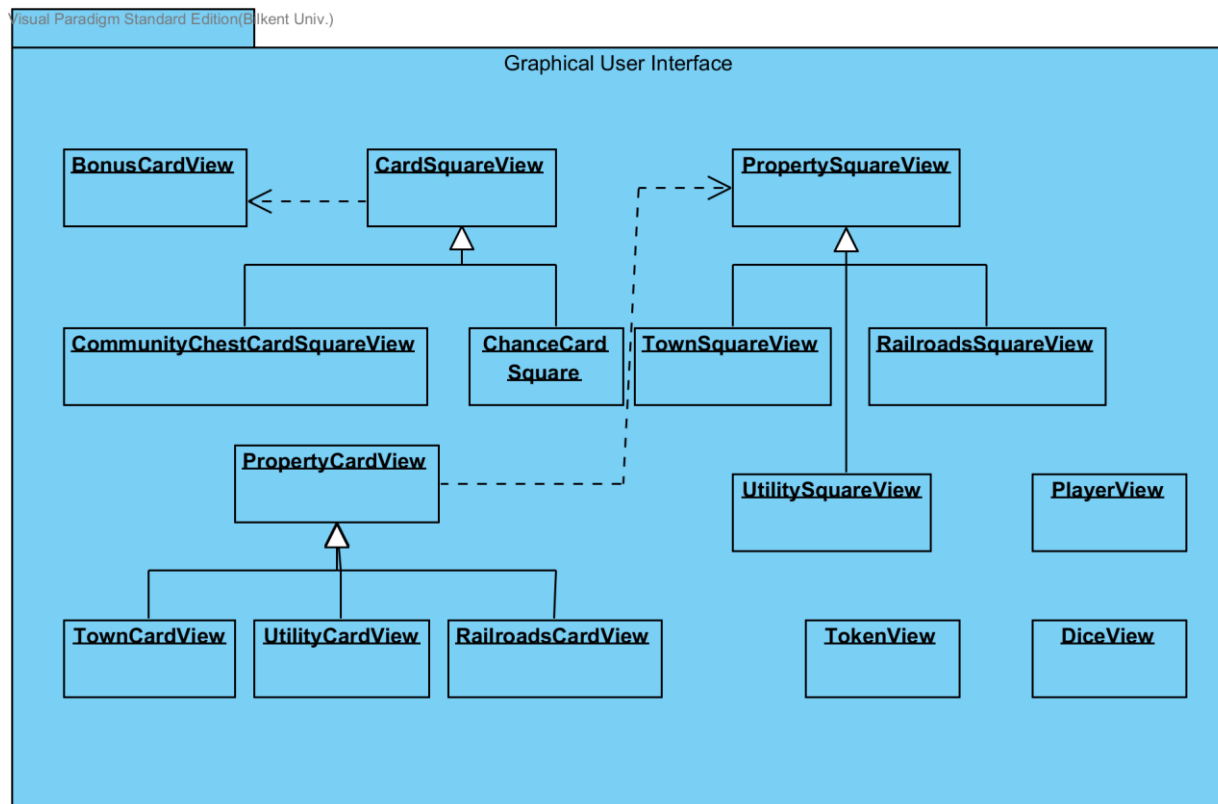


Figure 14 - Component diagram for GUI sub-system

4.2.2. Controllers Sub-system

This contains the controllers of the respective application object models and it contains a sub-package for database management and access:

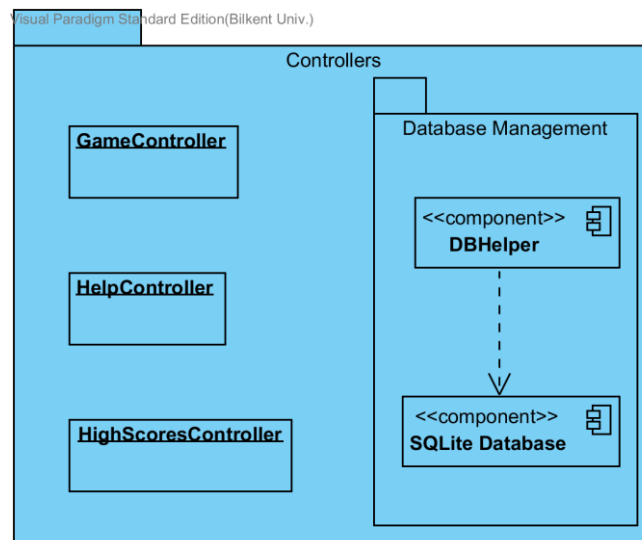


Figure 15 - Component diagram for Controllers sub-system

4.2.3. Models Sub-system

The following is the Models subsystem which contains all the model classes in the lowest layer of the architecture. It provides a façade with the MonopolyBoard class which also contributes to the strong coupling of the system.

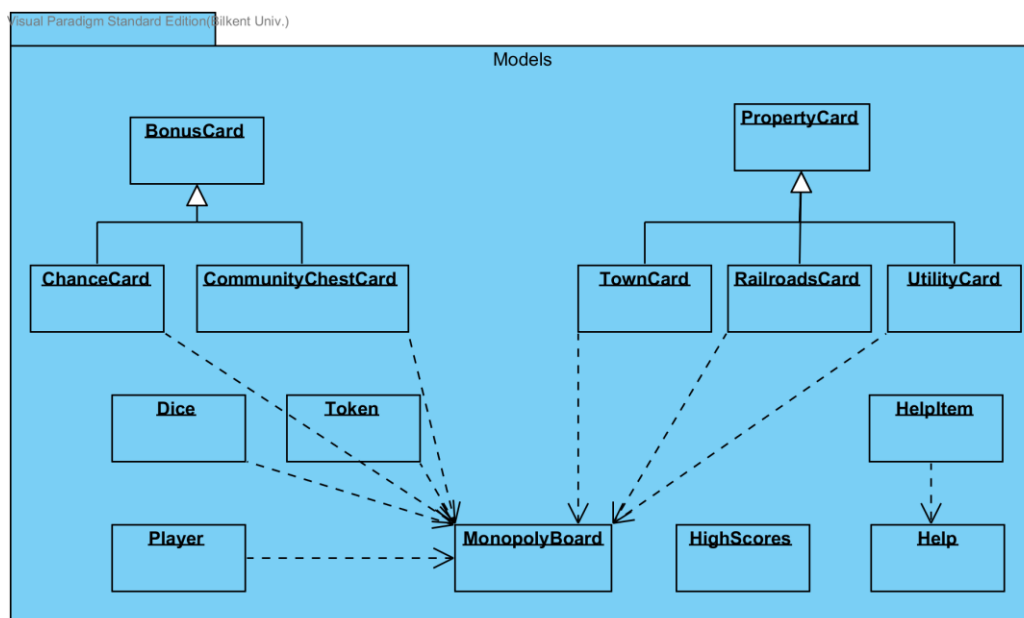


Figure 16 - Component diagram for Models sub-system

4.3. Architectural Patterns

The system architecture will be designed by following two patterns: 3-tier and model-view-controller (MVC). The system will have both hierarchical and functional separations amongst the classes, and these possible issues will be addressed by these architectural choices respectively.

The design process of the system consists of 3 parts: the examination, adaptation and also the evaluation of these options. In the process, both 3-tier and MVC architectures provide to be beneficial, however, the MVC design is the best support for our software.

In the system, the 3- tier architecture gives us the opportunities to easily maintain and upgrade any sub-systems independently in response to the changes in requirements, and the MVC patterns gives us flexibility in implementing the user interface by dividing a given software application into 3 interconnected components and separating the information into internal representations. Thus, the working mechanisms of these architecture designs will be helpful in the system that has many tightly bound classes.

4.3.1. Model-View-Controller (MVC)

The motivation and possible issues

The system will consist of multiple objects that will be used in the game logic, the controlling mechanism and the interaction aspect with the user realize the application itself. Since the relations and dependencies between all of the components may cause strong effects in any change that is made in any part of the system, some significant problems can be faced. One important problem can be that the user interface may be tend to change more frequently than the data storage system. In addition, since high coupling can make the classes difficult or impossible to reuse (they depend on a number of other classes), or applications tend to embody the logic that goes far beyond data transmission, it can be seen as another problem.

The addition of new data views may require re-implementation, refactoring of the business logic code, which then requires respective alterations in the other parts. In particular, considering the game μ Monopoly, we can be reassured that there is going to be a complex network of entities with their own operations and multiple attributes, and to be able to properly maintain the relationship among these agents and the environment, we chose the MVC architectural pattern.

The description of the MVC pattern

The Model-View-Controller (MVC) architectural design pattern solves these kind of problems by the separation method of the reflection of the domain (models), the presentation to the user (views) and the engine analyzing user interactions (controllers). In the system design, μ Monopoly domain objects such as tokens, squares, cards are the models that is to represent the domain of specific objects that must be modified and controlled. So, a controller part of the architecture, which the modifications of the controller is not classes forming the game logic and not directly controlled by themselves or by the user, should be necessary. In addition, the model objects can store a lot of important information which is needed by both view classes and controller. In the

architecture, the controller part is the game controller class which creates the game logic, with a certain hierarchy.

The view part manages the display of the interactions between the system and the user. These interactions are handled by view classes; which consist of all the screen, panel and label elements. Although the view part is mainly controlled and sends the interactions to the controller, it relates to the information from the models.

MVC can be seen in the game engines and applications where the view functions and the control functions are separated and the content of the game is chosen to be kept in the model part. Thus, the usage of the MVC design pattern in the system seems to be a good engineering practice.

4.3.2. Three-Tier Architectural Style

The system will have a hierarchical relationship among the relevant classes, and it is possible to create a framework for the logical design model, which can divide the components of the μ Monopoly application into three tiers of services corresponding to the logical layers of the application.

In the 3-tier architecture design pattern, the presentation layer displays the data to the user by permitting the manipulation and the entry of data through the controlled mechanisms.

The other component of this architecture is the middle layer that is made up of data and business rules. This is where the business logic is solved and the productivity advantages can be achieved.

The third part of the design pattern, is responsible for the storage of data in the database. Since the information for high scores and rules needs to be stored in μ Monopoly game, this should be an independent layer for the access of DBMS.

In this software, the three-tier architectural style provides many benefits such as usability, flexibility, reliability, maintainability, and adaptability, which are the some of the design goals of the our game. The services and the components can be shared and reused and in addition, they can be distributed across the data network. Thus, although the 3-tier pattern has its own advantages and can affect the resulting architecture choice, this design pattern has a deep importance for separation of components of the μ Monopoly game and reutilization of these components.

4.4. Hardware / Software Mapping

μ Monopoly is developed using the Java programming language thus it requires Java Runtime Environment in the running environment. This assures portability and due to the mixed interpretation / just-in-time compilation property it is platform-independent. In addition μ Monopoly is a software which does not communicated with exterior nodes through a network, thus as a standalone entity it will only depend on the installed, locally available components.

The following deployment diagram is a depiction of the component interaction and dependencies on the PC node. We will use Java Swing libraries for windowing and drawing GUI components, which uses AWT underneath, which further down uses the native drawing engine, but these details are abstracted here. Moreover, the system will use a SQLite database which is a separate entity to which we communicate via SQL language and the Java Persistence APIs.

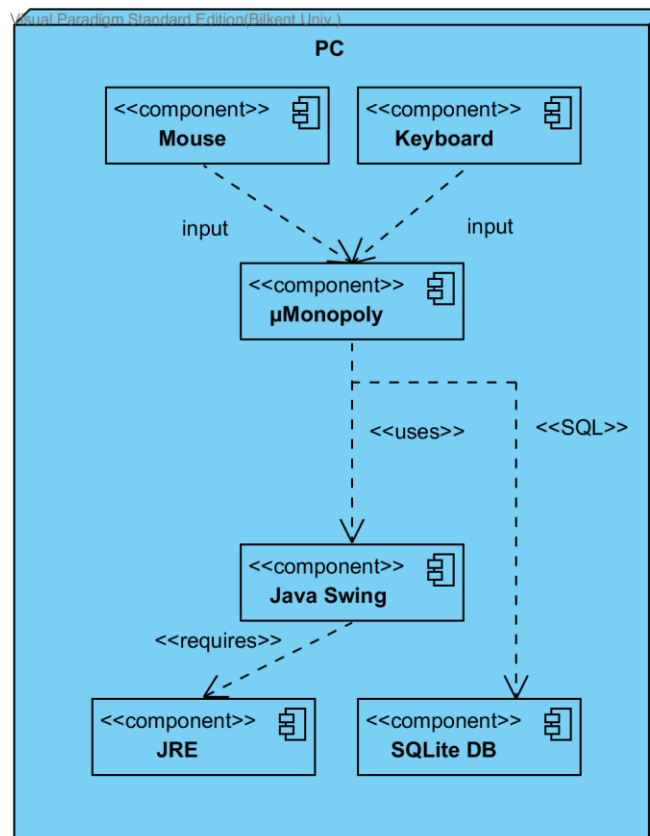


Figure 17 - Deployment diagram for Hardware-Software mapping

4.5. Addressing Key Concerns

4.5.1. Persistent Data Management

The game will store High Scores and Rules. For that purpose we will utilize a database and namely SQLite, which is a relational database. SQLite is selected because it takes up less space in the hard drive and it is relatively easy to use by the Java Persistence API. The JAR library called sqlite-jdbc-3.7.2 is used for running the database in our Java application.

There will be two tables in our DB: The High Scores table, which saves the names of the users who won the game, tokens they chose, amount of money that they had upon winning, the date and time of winning as well as their assigned IDs. These IDs will be used as primary keys to manage data records in the database. The data will be showed on the High Score screen as sorted by the highest amount of money that the user had at the end of the game. In addition the other information kept in the columns described above will be displayed as well.

On the other hand, the Rules table will keep persistent information on the game as for example the card names, card properties (rent, sell, mortgage, build values etc.), Change and Community Chest cards names and contents – optionally icons.

4.5.2. Access Control and Security

Access control is the security technique for the regulation of which actors can access certain classes and methods. Access control is composed of different and numerable functionality and data that can be accessed by various entities. Since the μ Monopoly game has only one actor there is no such an object guard system and the user will not be authenticated, in contrary, they will be able to access the entire set of functionalities of the game as provided.

4.5.3. Global Software Control

There are three kinds of software control mechanisms, procedure-driven control, event driven control and threads, but in our software, we will use event-driven control. In an Event-driven mechanism, the events and event handlers dictate flow of the system as each handler responds to a particular stimuli coming from an event trigger. We use Model View Controller (MVC) design pattern in our μ Monopoly game, therefore, in collaboration with Java Swing, it is more suitable to event-driven mechanism. On the other hand, this is compatible to having a graphical user interface, where a user interaction to the interface corresponds to an event delegated to the Controller classes such as GameController which will then ask the models for a change or update while this is reflected again back to the interface.

4.5.4. Boundary Conditions

Initialization

- The game starts as the user opens the file.
- The program is loaded into memory and becomes ready to use.
- The JVM loads the main screen's view and the rest of model, controller and view classes.
- The graphics are drawn using Java's Swing library.
- The main screen is displayed where there are four buttons to be selected by users: Start Game, High Scores, Help and Quit.
- If the Start Game is selected, the user is redirected to create the profiles and then start the game.
- If High Scores is selected, the system will display the sorted list of high score records obtained from the database.
- If Help is selected, the system will display a new window with the help content.
- Finally, if Quit is selected the application will terminate.

Termination

- Users can terminate the program whenever they want to.
- The termination can be performed by clicking on the usual red exit button in the top-right of the frame for Windows OS and top-left for Mac OS, or by clicking the Quit buttons in the Home Screen or Game Screen.
- If the game is still continuing, just after the request for termination is done, the system, however, asks the user whether they really want to quit. In case they decide to quit the records will not be saved.
- Only in the case when the game has finished the records of the winning player will be added automatically to the DB.
- All the data that is not saved in database, is deleted upon exiting and memory is freed.

Failure

- If the user does not have the JRE installed in the system, the executable file won't run and this cannot be solved unless they install Java.
- If the user tries to force exit while the game is running or it has just finished, the scores of the played game may not be saved into database and the effect is not recoverable.
- If the user tries to run the game while one instance is already running in the JRE at the same time, there might be some database errors like "database is busy" or "database is locked". These reassure that the database will be safe from the unsynchronized editing attempts and the data won't be lost.
- In cases of power cut or sudden termination of the system environment the game will not be able to recover the progress made, nor the scores of each player. However this will not affect the database content.

5. Object Design

5.1. Pattern Application

5.1.1. Abstract Factory Pattern

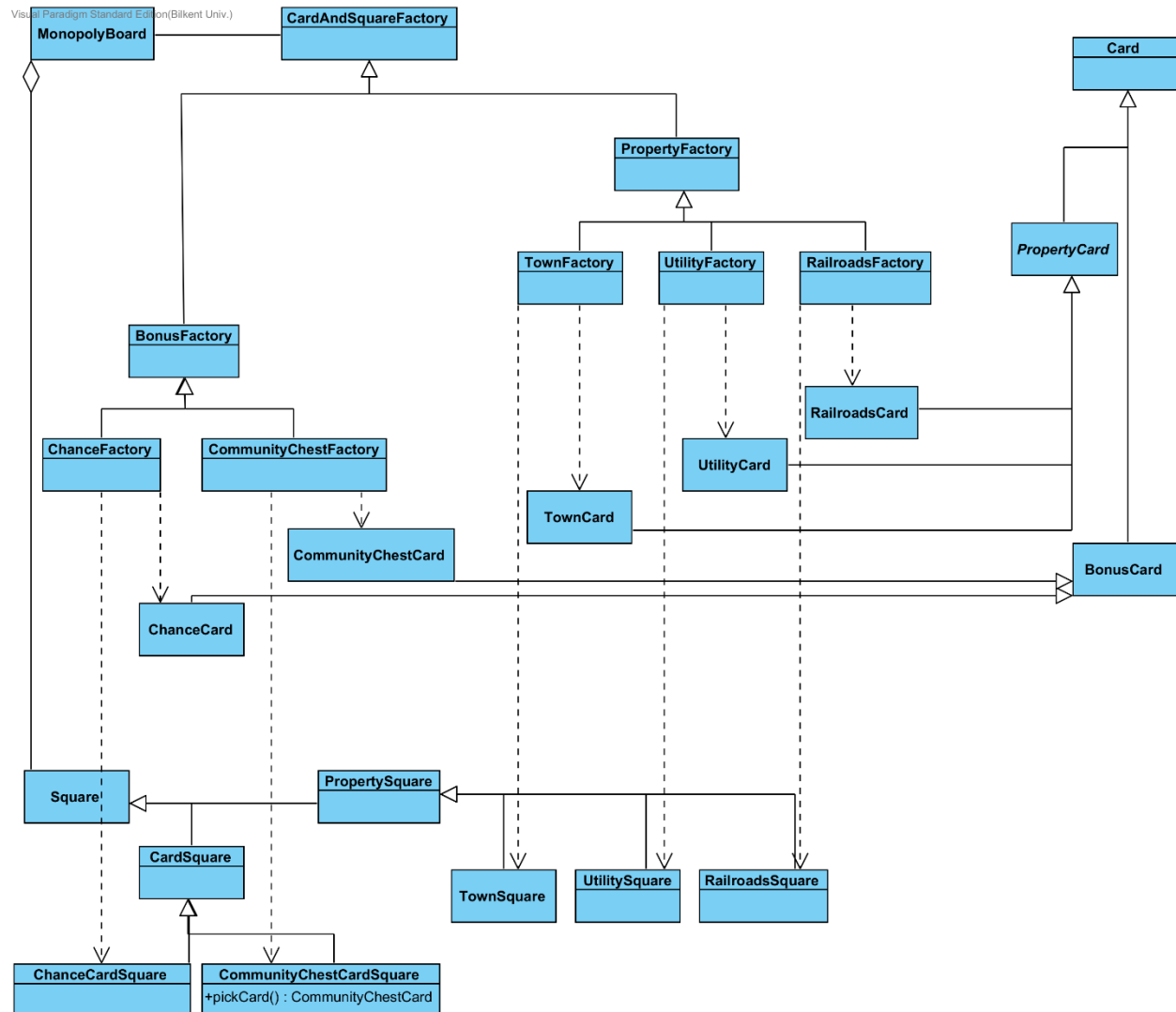


Figure 18 - Application of Abstract Factory Pattern

The diagram above draws the relevant classes to show the application of the Abstract Factory design pattern. The **CardAndSquareFactory** is an umbrella class whose two children are the actual Abstract Factory implementations: **BonusFactory** and **PropertyFactory**. These factories are further extended under **ChanceFactory**, **CommunityChestFactory**, **RailroadsFactory**, **UtilityFactory** and **TownFactory**, which are responsible for creating a bundle of "products" – cards and squares in this case. **Card** and **Square** are thus abstract products and each square is associated with a particular type of card according to the inheritance relationship presented in the UML diagram in Section 3.1.2.

The uMonopoly board is made up of 40 squares which have completely different behavior and are classified into major groups as shown above. This is a detailed specialization of the classes that provides greater abstraction of implementation and is useful to maintain a genuine OOP paradigm, however this may give rise to the problem of missing and confused ties between the squares and the cards and results in a wrongly connected system. To address this issue we adopted the Abstract Factory design pattern and introduced extra factory classes with the benefit of a tightly structured network of classes.

5.1.2. Builder Pattern

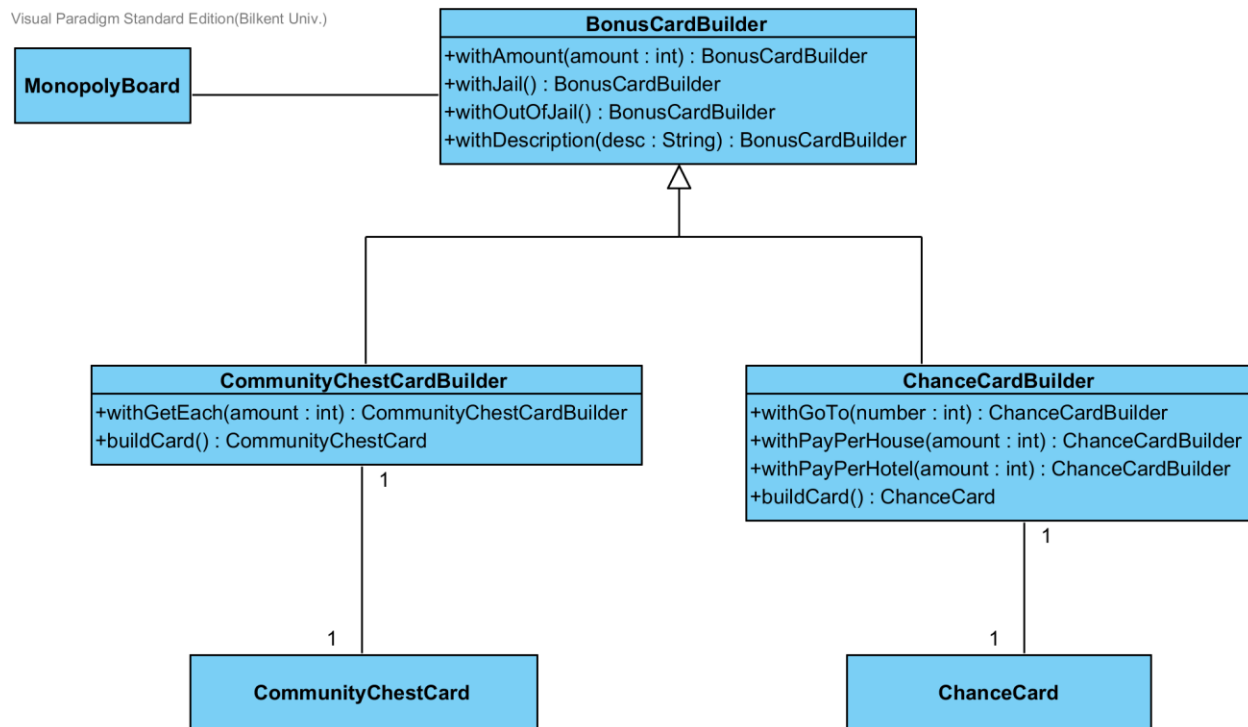


Figure 19 - Application of the Builder Pattern

The Builder design pattern is another one that we adopted with the intention of creating BonusCard objects. These can either be ChanceCard or CommunityChestCard objects and they contain multiple properties that describe their behavior such as the amount of money to transfer, the method of transfer, the displacement of the player etc. However not every card will assign such values and besides the description the other attributes are all optional. This would lead us to have a number of constructors where one attribute is added in each – a cumbersome and relatively bad design. This issue is best solved by applying the Builder pattern so that for adding the properties the builder's provided methods are used and the ones left out assign to the default values.

5.1.3. Observer Pattern

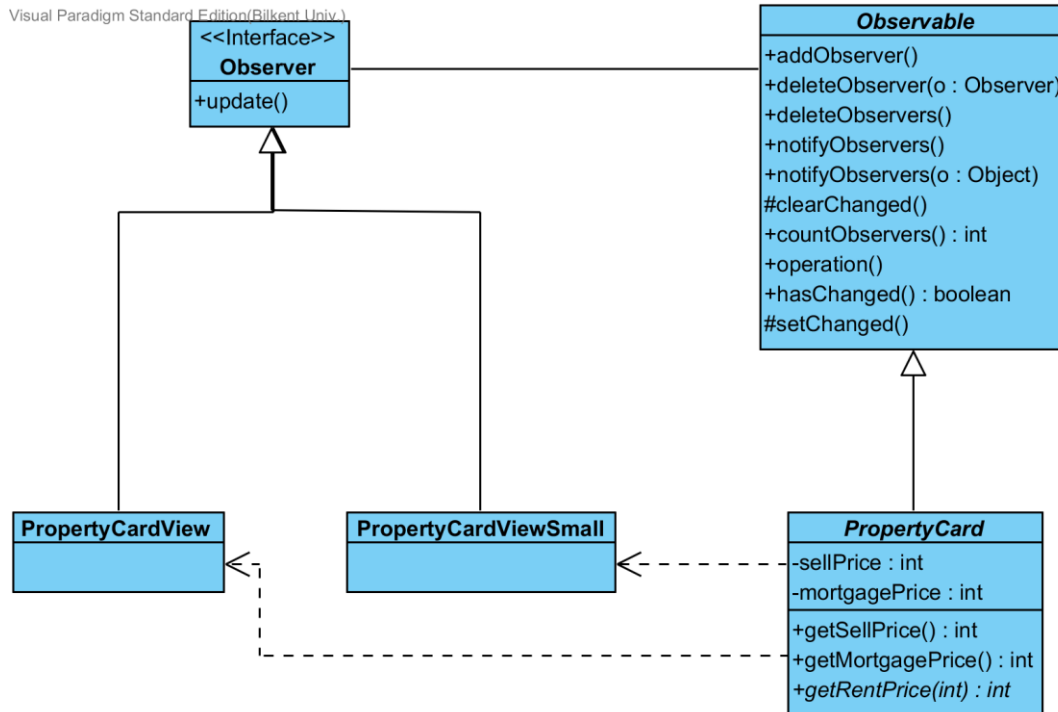


Figure 20 - Application of Observer Design Pattern

Our project is built using the MVC architectural pattern that separates the data objects from the boundary ones and thus allows for independence and decoupling. This situation leads for weak communication and unsynchronized data, which can be resolved by applying the Observer pattern as displayed in the diagram above. One example of this Model-View tie using the Observer design pattern is shown as the Dice and the DiceView by using the Java API classes `Observer` and `Observable`. The `DiceView` class is able to subscribe to the `Dice` model by using the methods of the `Observable` class such as `addObserver(Observer o)` and passing the self-reference (`this`). On the other hand, when the model changes the internal data it can let the `DiceView` know by calling the `notifyObservers()` method which iterates through the list of subscribers (observers in this case) and calls their `update` method. This is perfectly fitting with our aspiration of implementing an Object-Oriented method of Observer design pattern and we applied this to all the models which have a view.

5.1.4. Façade Pattern

The Façade pattern is implemented in smaller scale as compared to the design patterns above and it consists of the `MonopolyBoard` class which plays the façade role for the Domain subsystem shown in Figure 15. In this way it covers the details behind the system by wrapping all the models in one concise entity and providing services that are natural for the interacting objects but underneath distributing the work onto the covering models and controllers.

5.1.5. Composite Pattern

The Composite pattern is indirectly implemented from our project since we are using the Java Swing library to draw the user interface and it organizes its children by means of the Composite structural design pattern. An example would include the fact that in a JPanel there can be different components such as JLabel and JButton but it can also contain other JPanels which recursively will give a variable depth to the structure. Additionally, Java Swing uses the AWT Toolkit underneath and it also contains the same structure hierarchy.

5.2. Class Interfaces

Dice
- value1 : int - value2 : int - type : DiceType
+ Dice() + rollAndGetTotalValue() : int - roll(): void + isDoubles: boolean + getValue2():int + getValue1():int + getType(): DiceType + setType(): void

Dice

This is a simulation of the two dice used in the Monopoly Game. It provides the rolling method tailored for the different Dice Types.

<<enum>> DiceType
+ <u>GOLDEN</u> + <u>PLATINUM</u> + <u>SIMPLE</u>

DiceType

This is enumeration that defines the dice types.

Token
<ul style="list-style-type: none"> - position : int - figure: TokenFigure - type : TokenType
<ul style="list-style-type: none"> + Token(figure : TokenFigure) + equals(): boolean + getType: TokenType + setType(): void + getFigure(): TokenFigure + getPosition(): int + setPosition(): void

Token

This is simulation of tokens used in the game. They keep track of the position (the square in which they are landed) and they have a figure and a type.

<<enum>> TokenFigure
<ul style="list-style-type: none"> - icon: ImageIcon - selectedIcon: ImageIcon + <u>DOG</u> + <u>CAR</u> + <u>HAT</u> + <u>THIMBLE</u> + <u>SHOE</u> + <u>IRON</u>
<ul style="list-style-type: none"> - TokenFigure(pathToIcon : String, pathToSelectedIcon : String) + getIcon(): ImageIcon + getSelectedIcon(): ImageIcon

TokenFigure

This is an enumeration type that represents the visual display of a token.

<<enum>> TokenType
<ul style="list-style-type: none"> - <u>GOLDEN</u> - <u>PLATINUM</u> - <u>SIMPLE</u>

TokenType

This is an enumeration type that represents the type of the Token upgrade.

<i>BonusCard</i>
<ul style="list-style-type: none"> - id : int - amount : int - jail : boolean - outOfJail : boolean - desc : String
<pre># BonusCard(amount : int, jail : boolean, outOfJail : boolean, desc : String) + getPayAmount() + getID() + isJailCard() + isOutOfJailCard() + getDescription()</pre>

BonusCard

This class is an abstract class that provides the necessary properties and the methods for the Chance and Community Chest cards without no implementation.

<i>ChanceCard</i>
<ul style="list-style-type: none"> - goTo : Integer - payPerHouse : int - payPerHotel : int - view : ChanceCardView
<pre>+ ChanceCard(amount : int, jail : boolean, outOfJail : boolean, desc : String, goTo: Integer, payPerHouse : int, payPerHotel : int) + getGoToDestination() + getPayPerHouseAmount() + getPayPerHotelAmount()</pre>

ChanceCard

Extends: BonusCard

This class uses all of the properties and methods of BonusCard by extending the BonusCard class. According to the descriptions of the chance cards, the user can go to the place written on the card or can pay amount for per house and per hotel in Monopoly game.

CommunityChestCard
- getEach: int - view : CommunityChestCardView
+ CommunityChestCard(amount : int, jail : boolean, outOfJail : boolean, desc : String, getEach: int) + getPayAmountForEachPlayer()

CommunityChestCard

Extends: BonusCard

This class is to represent the community chest cards in the game and they are specialized cards similar to the bonus cards. The user can pay amount for each player according to the explanations on the community chest cards.

PropertyCard
- sellPrice : int - mortgagePrice : int - rentPrice : int[]
PropertyCard(sellPrice : int, mortgagePrice :int) + getSellPrice() + getMortgagePrice() # getRentPrices() # setRentPrices(rents : int[]) + getRentPrice(number : int)

PropertyCard

This class that is an abstract class is to represent the property cards in the game. Each property card represents the places written in the monopoly board and the places are divided based on their colors. The user can have a one or more property by paying the amount based on the explanations on the card. Also, selling price, mortgage price and rent price are written on the property cards.

RailroadsCard
- view : RailroadsCardView
+ RailroadsCard(sellPrice : int, mortgagePrice : int, rent1 : int , rent2 : int, rent3 : int, rent4 : int) + getRentPrice(nrOfStations : int)

RailroadsCard

Extends: PropertyCard

This class is a representation of the railroads in the monopoly game. The railroads have the highest possibility to be visited by the players and there are 4 railroads in the monopoly board. On the each railroad part, the rent price are written.

TownCard
<ul style="list-style-type: none"> - housePrice : int - hotelPrice : int - view : TownCardView
<ul style="list-style-type: none"> + TownCard(sellPrice: int, mortgagePrice : int, rent0 : int , rent1 : int, rent2: int, rent3 : int, rent4 : int, rentHotel : int) + getRentPrice(nrOfHouses : int) + getHotelRentPrice() + getHouseBuildPrice() + getHotelBuildPrice()

TownCard

Extends: PropertyCard

This class is to demonstrate the town cards in the monopoly game. TownCard class can benefit from the all methods and properties from PropertyCard class. The rent and build prices of hotels and houses can be obtained from the TownCard class.

UtilityCard
<ul style="list-style-type: none"> - view : UtilityCardView
<ul style="list-style-type: none"> + UtilityCard(sellPrice : int, mortgagePrice : int, rent1 : int, rent2 : int) + getRentPrice(nrOfUtilities : int)

UtilityCard

Extends: PropertyCard

This class is a demonstration of utility cards in the monopoly game. There are 2 types of utility cards, the electric company and water works. Any building cannot be placed on a utility and if a player owns a utility, the rent prices should be obtained so that the rent can be changed according to the description on the UtilityCard.

Help
<ul style="list-style-type: none"> - helpItems : List<HelpItem> - helpView : HelpView
<ul style="list-style-type: none"> + Help(view : HelpView, items : HelpItem[]) + getItemAt(index : int)

Help

The Help class is the collector of the help items, and it keeps track of the currently displayed HelpItem by providing extra navigation methods.

HelpController
- currentItem : int - helpModel : Help
+ HelpController(helpModel : Help) + getCurrentItem() + getPreviousItem() + getNextItem() + getItemAt()

HelpController

This class controls the user interaction with the Help class by delegating the method calls from the GUI input.

HelpItem
- title : String - description : String
+ HelpItem(title : String, description : String) + getTitle() + getDescription()

HelpItem

The HelpItem class has the title and description and the respective methods of a singular Item that is displayed on the Help screen.

CardSquare
+ pickCard()

CardSquare

This class is an abstract specialization of the Square class and it holds the pickCard() method. The reason for not doing this an interface is related to the implementation of the Abstract Factory design pattern. (Refer section 5.1.1)

ChanceCardSquare
<ul style="list-style-type: none"> - theChanceCardList : List<ChanceCard> - selectedCardID : int - maxSelectableCardSize : int - selectableCardSize : int
<ul style="list-style-type: none"> + ChanceCardSquare(position : int, name : String, type : SquareType, theChanceCardList : List<ChanceCard>) + pickCard()

ChanceCardSquare

Extends: CardSquare

This class holds a list of chance cards and has a method pickCard() which selects a random chance card.

CommunityChestCardSquare
<ul style="list-style-type: none"> - theCommunityChestCard : List<CommunityChestCard> - selectedCardID : int - maxSelectableCardSize : int - selectableCardSize : int
<ul style="list-style-type: none"> + CommunityChestCardSquare(position : int, name : String, type : SquareType, theCommunityChestcard : List<CommunityChestCard>) + pickCard()

CommunityChestCardSquare

Extends: CardSquare

This class hold a list of community chest cards and has a method pickCard() which selects a random community chest card.

PropertyGroup
<ul style="list-style-type: none"> - type : PropertyGroupType - properties : List<PropertySquare> - players : List<Player>
<ul style="list-style-type: none"> + PropertyGroup(type : PropertyGroupType, properties : List<PropertySquare>) + ownsAllProperties(thePlayer : Player) + getType()

PropertyGroup

This class holds the list of players and the list of property squares and has a method ownsAllProperties() that checks whether a player has properties with the same color or not. If the player has the properties with same color, he/she can build a house.

<<enum>> PropertyGroupType
+ <u>RAILROADS</u> + <u>TOWN</u> + <u>UTILITY</u>

PropertyGroupType

This class is an enum class that only holds RAILROADS, TOWN and UTILITY respectively. It helps the GameController know if the user can build in a town and the calculation of the rent amount according the number of owned properties.

<i>PropertySquare</i>
+ PropertySquare(position : int, name : String, type : SquareType) + <i>getRentPrice(numberOfBuilding : int)</i>

PropertySquare

Extends: Square

This class is an abstract class which has the methods and properties of the Square class but specializes in order to provide rent prices and wrap all the 'ownable' squares.

RailroadsSquare
- theRailRoadsCard : RailRoadsCard
+ RailroadsSquare(position : int, name : String, type : SquareType, theRailroadsCard : RailroadsCard) + getRentPrice(numberOfBuildings : int)

RailroadsSquare

Extends: PropertySquare

This class has the properties and the methods of PropertySquare and represents the rail road squares.

Square
<u>- idCounter : int</u> # id : int # position : int # name : String # type : SquareType
+ Square(position : int, name : String, type : SquareType) + getID() + getName() + getSquareType() + getPosition()

Square

This is the main class that represents any square in the Monopoly board game.

SquareType
+ <u>PROPERTY</u> + <u>CHANCE</u> + <u>COMMUNITY_CHEST</u> + <u>GO</u> + <u>JAIL_VISITOR</u> + <u>JAIL</u> + <u>FREE_PARKING</u> + <u>TAX</u>

SquareType

This class is an enum class that only holds PROPERTY, CHANCE, COMMUNITY_CHEST, GO, JAIL_VISITOR, JAIL, FREE_PARKING, TAX to represent the types of the squares in the Monopoly board respectively.

TownSquare
- color : Color + houseCount : int + hotel : boolean - theTownCard : TownCard
+ TownSquare(position : int, name : String, type : SquareType, color : Color, houseCount : int, hotel : boolean, theTownCard : TownCard) + addHouse() + addHotel() + getColor() + getHouseCount() + hasHotel() + getRentPrice(numberOfBuilding : int)

TownSquare

Extends: PropertySquare

This class represents the towns written in the Monopoly game and has all the methods and properties of PropertySquare. Besides, the TownSquare class has additional methods for adding house or hotel, getting the color of the properties, getting the number of the houses, checking whether there is a hotel or not and getting rent prices.

UtilitySquare
- theUtilityCard : UtilityCard
+ UtilitySquare(position : int, name : String, type : SquareType, theUtilityCard : UtilityCard)
+ getRentPrice(numberOfBuildings : int)

UtilitySquare

Extends: PropertySquare

This class has all of the methods and properties of PropertySquare and represents the squares of utilities.

DatabaseHelper
- theHighScores : ArrayList<HighScores>
- dbConnection: Connection
- stmt: Statement
+ DataBaseHelper()
+ getHighScoreFromDB(): ArrayList<HighScores>
+ insertHighScoreToDB(): void
+ removeHighScoreFromDB(): void
- sortingHighScores(): void

DatabaseHelper

DatabaseHelper provides the connection class between the application and μMonopoly's SQLite database. This class gets the high scores from the database and sorts them. Additionally it allow for entering new HighScore records and deleting selected High Score records or simply all of them.

HighScores
<ul style="list-style-type: none"> - id: int - name: String - tokenFigure: TokenFigure - amount: int - date: String
<ul style="list-style-type: none"> + HighScores(id:int, name:String, tokenFigure:String, amount:int, date:String) + getId():int + getName():String + getTokenfigure():TokenFigure + getAmount():int + getDate():String

HighScores

This is the respective data object for the database records about the information of the players with the highest scores, their names and token figures.

Rules
<ul style="list-style-type: none"> + <u>MAX_PLAYERS</u>: int + <u>MIN_PLAYERS</u>: int + <u>MAX_HOUSE_COUNT</u>: int + <u>STARTING_BUDGET</u> : int + <u>SALARY</u> : int + <u>TAX_INCOME</u> : int + <u>TAX_LUXURY</u> : int + <u>SQUARE_ORDER</u> : int + <u>CHANCE_CARDS_COUNT</u> : int + <u>COMMUNITY_CARDS_COUNT</u>: int

Rules

This class will keep the static constant information about the rules aspects such as described in the official playing guide of the Monopoly game.

ChanceCardView
<ul style="list-style-type: none"> - mainPanel : JPanel - descPane: JTextPane
+ ChanceCardView()

ChanceCardView

The view representing a Chance Card.

ChanceSquareView
<ul style="list-style-type: none"> - mainPanel : JPanel - firstTokenLb: JPanel - secondTokenLb: JPanel - thirdTokenLb: JPanel - fourthTokenLb: JPanel
<ul style="list-style-type: none"> + getContent():JPanel + addTokenFigure() + removeTokenFigure()

ChanceSquareView

The view representing a Chance Square.

CommunityChestCardView
<ul style="list-style-type: none"> - mainpanel: JPanel - descPane: JTextPane
+ CommunityChestCardView()

CommunityChestCardView

The view representing a Community Chest Card.

CommunityChestSquareView
<ul style="list-style-type: none"> - mainpanel: JPanel - tileLb: JPanel - iconLb: JPanel - firstTokenLb: JPanel - secondTokenLb: JPanel - thirdTokenLb: JPanel - fourthTokenLb: JPanel
<ul style="list-style-type: none"> + CommunityChanceSquareView(square: CommunityChestCardSquare) + getContent():JPanel + addTokenFigure() + removeTokenFigure()

CommunityChestCardSquareView

The view representation of a Community Chest Card Square.

CornerSquareView
<ul style="list-style-type: none"> - mainpanel: JPanel - iconLb: JPanel - firstTokenLb: JPanel - secondTokenLb: JPanel - thirdTokenLb: JPanel - fourthTokenLb: JPanel
<ul style="list-style-type: none"> + getContent():JPanel + addTokenFigure() + removeTokenFigure()

CornerSquareView

The view representation of the Corner Squares.

DiceView
<ul style="list-style-type: none"> - mainPn : JPanel - firstDieLb : JLabel - secondDieLb : JLabel - <u>ONE</u> : ImageIcon - <u>TWO</u> : ImageIcon - <u>THREE</u> : ImageIcon - <u>FOUR</u> : ImageIcon - <u>FIVE</u> : ImageIcon - <u>SIX</u> : ImageIcon
<ul style="list-style-type: none"> + getContent() : JPanel

DiceView

The view representation of the dice with all the faces represented with the traditional interface with dots.

HomeScreen
<ul style="list-style-type: none"> - content:JPanel - start:JButton - highScores: JButton - help: JButton - quit: JButton - title:JLabel - credits: JLabel - buttons: JPanel
<ul style="list-style-type: none"> + HomeScreen() + getContent(): JPanel

HomeScreen

This is the first landing screen as soon as the application is run. It give the user four choices from the main menu: Start Game, View High scores, View Help and Quit.

PlayerRegistrationScreen
<ul style="list-style-type: none">- playerRegSections: PlayerRegistrationSection[]- mainPanel: JPanel- backBtn: JButton- nextBtn: List<PlayerRegistrationSection>- mainPanel:JPanel- backBtn: JButton- nextBtn: JButton- northPn: JPanel- titleLb: JLabel- subTitleLb: JLabel- centerPn: JPanel- southPn: JPanel- plusBtn: JButton- shownPlayerSections: int
<ul style="list-style-type: none">+ PlayerRegistrationScreen(gameController: GameController)+ update()+ checkTokens()+ getContent():JPanel

PlayerRegistrationScreen

This is the screen presented as soon as the user clicks Start Game form the main menu. It contains the PlayerRegistrationSection objects and provides back and next navigations in addition.

PlayerRegistrationSection
<ul style="list-style-type: none">- <u>NO_TOKEN_SELECTED</u>: String- <u>NO_NAME_ENTERED</u>: String- <u>NAME_NOT_UNIQUE</u>: String- <u>FEW_PLAYERS</u>: String- mainpanel: JPanel- nameTf: JTextField- nameLb: JLabel- dogBtn: JButton- carBtn: JButton- shoeBtn: JButton- thimbleBtn: JButton- hatBtn: JButton- ironBtn: JButton

- namePn: JPanel - playerNrLb: JLabel - content: JPanel - errorLb: JLabel - isLocked:boolean
+ PlayerRegistrationSection(playerNr:int) + getName():String + getTokenFigure():TokenFigure + setError() + setVisible() + disableTokens() + lock()

PlayerRegistrationSection

This is one particular row in the PlayerRegistrationScreen. It allows the user to register the players with unique names and tokens in order to proceed to the main game screen.

PropertyCardSmallView
- mainPanel: JPanel - titleLb:JLabel
+ PropertyCardSmallView(card:PropertyCard) + getContent(): JPanel

PropertyCardSmallView

The representation of the Property Card to be shown on the Sidebar of the main game screen. It will give the player the information on which cards he / she owns.

RailroadsCardView
- mainPanel: JPanel - titleLb:JLabel - rentValueLb:JLabel - twoStationsRentLb: JLabel - threeStationsRentLb: JLabel - fourStationsRentLb: JLabel - mortgageValueLb: JLabel
+ RailroadsCardView(card:RailroadCards) + getContents():JPanel

RailroadsCardView

A view representation of the RailroadsCards.

RailroadsSquareView
<ul style="list-style-type: none"> - mainPanel: JPanel - titleLb: JLabel - firstTokenLb: JLabel - secondTokenLb: JLabel - thirdTokenLb: JLabel - fourthTokenLb: JLabel
<ul style="list-style-type: none"> + RailroadsSquareView(square:RailroadsSquare) + getContent(): JPanel + addTokenFigure() + removeTokenFigure()

RailroadsSquareView

A view representation of the RailroadsSquare.

TaxSquareView
<ul style="list-style-type: none"> - mainPanel: JPanel - titleLb: JLabel - firstTokenLb: JLabel - secondTokenLb: JLabel - thirdTokenLb: JLabel - fourthTokenLb: JLabel
<ul style="list-style-type: none"> + TaxSquareView(square:TaxSquare) + getContent():JPanel + addTokenFigure() + removeTokenFigure()

TaxSquareView

A view representation of the tax squares.

TownCardView
<ul style="list-style-type: none"> - mainPanel: JPanel - titlePn:JPanel - firstRentLb:JLabel - rentPn:JPanel - titleLb: JLabel - oneHouseLb: JLabel - twoHousesLb: JLabel - oneHouseRentLb: JLabel - twoHousesRentLb: JLabel - threeHousesLb: JLabel - threeHousesRentLb: JLabel - fourHousesLb: JLabel - fourHousesRentLb: JLabel

<ul style="list-style-type: none"> - hotelLb: JLabel - hotelLb: JLabel - hotelRentLb: JLabel - mortgageValueLb: JLabel - houseCostLb: JLabel - hotelCostLb: JLabel
<ul style="list-style-type: none"> + TownCardView(card:TownCard) + getContent():JPanel

TownCardView

A view representation of the Town Cards.

TownSquareView
<ul style="list-style-type: none"> - mainPanel: JPanel - nameLb:JLabel - colorPn:JPanel - firstHouseLb: JLabel - secondHouseLb: JLabel - thirdHouseLb: JLabel - fourthHouseLb: JLabel - tokenPn:JPanel - firstToken:JLabel - secondToken:JLabel - thirdToken:JLabel - fourthToken:JLabel - houseIcon:ImageIcon - hotelIcon:ImageIcon
<ul style="list-style-type: none"> + TownSquareView(square:TownSquare) + getContent():JPanel + addHouse + removeHouse + addTokenFigure() + removeTokenFigure()

TownSquareView

A view representation of the Town squares.

UtilityCardView
<ul style="list-style-type: none"> - pictureLb: JLabel - oneUtilityLb: JLabel - twoUtilityLb: JLabel - mortgageValueLb: JLabel - titleLb: JLabel - mainPanel: JPanel
<ul style="list-style-type: none"> + UtilityCardView(card:UtilityCard) + getContent: JPanel

UtilityCardView

A view representation of the Utility cards.

5.3. Specifying Contracts

TownSquare

1. **Context** TownSquare::addHouse() **pre**:
self.houseCount < Rules.MAX_HOUSE_COUNT **and** self.hotel == false

2. **Context** TownSquare::addHouse() **post**:
self.houseCount = self@pre.houseCount + 1

3. **Context** TownSquare::addHotel() **pre**:
self.houseCount == Rules.MAX_HOUSE_COUNT **and** self.hotel == false

4. **Context** TownSquare::addHotel() **post**:
self.hotel == true **and** self.houseCount == 0

DBHelper

5. **Context** DBHelper::getHighScoreFromDB() **post**:

self.dbCoonection.isClosed() **and** self.stmt.isClosed() **and**
self.dbConnection.isCommitted()

6. **Context** DBHelper::insertHighScoreToDB() **post**:

self.dbCoonection.isClosed() **and** self.stmt.isClosed() **and**
self.dbConnection.isCommitted()

Dice

7. **Context** Dice **inv**:

self.value1 > 0 and self.value1 < 7 **and** self.value2 > 0 and self.value2 < 7

8. **Context** Dice::rollAndGetTotalValue() **pre**:

self.type == DiceType.SIMPLE **or** type == DiceType.GOLDEN **or** type ==
DiceType.PLATINUM

9. **Context** Dice::rollAndGetTotalValue(DiceType nDice) **post**:

self.value1 != @pre.self.value1 **and** self.value2 != @pre. self.value2

Token

10. **Context** Token **inv**:

position >= 0 **and** position < 40

11. **Context** Token **inv**:

type == TokenType.SIMPLE **or** type == TokenType.PLATINUM **or** type ==
TokenType.GOLDEN

12. Context Token inv:

figure == TokenFigure.DOG **or** figure == TokenFigure.CAR **or** figure == TokenFigure.HAT
or figure == TokenFigure.THIMBLE **or** figure == TokenFigure.SHOE **or** figure ==
TokenFigure.IRON

Game

13. Context Game inv:

self.turn >= 0 **and** self.turn < 4

14. Context Game inv:

self.currentSquares.size == 4

15. Context Game inv:

self.players.size >1 and self.players.size <= 4

16. Context Game inv:

self.playerNames.size >1 **and** self.playerNames.size <= 4

17. Context Game inv:

self.playerNames.size == self.players.size

18. Context Game::addPlayer(String name, TokenFigure tokenFigure) pre:

not playerNames->includes (name)

19. Context Game::addPlayer(String name, TokenFigure tokenFigure) post:

self.players.size == self@pre.players.size + 1

20. Context Game::addPlayer(String name, TokenFigure tokenFigure) **post:**

self.playerNames.size == self@pre.playerNames.size + 1

21. Context Game::removePlayer(String name) **pre:**

self.players.size > 0

22. Context Game::removePlayer(String name) **pre:**

self.playerNames -> includes(name)

23. Context Game::removePlayer(String name) **post:**

self.players.size == self@pre.players.size - 1

24. Context Game::removePlayer(String name) **post:**

self.playerNames.size == self@pre.playerNames.size - 1

Game Controller

25. Context GameController::StartGame() **pre:**

self.game == null

26. Context GameController::buyProperty() **pre:**

game.getPlayer(currentPlayer).getBudget() >= game.getSquare(currentSquare).getPrice()

27. Context GameController::buyProperty() **pre:**

not game.getPlayer(currentPlayer).ownsProperty(game.getSquare(currentSquare))

28. Context GameController::sellProperty() **pre:**

game.getPlayer(currentPlayer).ownsProperty(game.getSquare(currentSquare))

29. Context GameController::sellProperty() **post:**

not game.getPlayer(currentPlayer).ownsProperty(game.getSquare(currentSquare))

30. Context GameController::startGame () **post:**

self.game != null

MonopolyBoard

31. Context MonopolyBoard **inv:**

self.squareList.size == 40

32. Context MonopolyBoard::build(int currentSquare) **pre:**

self.squareList -> at(currentSquare).getHouseCount() < 5 **or** (self.squareList -> at(currentSquare).getHouseCount() == 4 **and not** self.squareList -> at(currentSquare).hasHotel())

33. Context MonopolyBoard::build(int currentSquare) **post:**

self.squareList -> at(currentSquare).getHouseCount() == self@pre.squareList -> at(currentSquare).getHouseCount() + 1 **or** (self.squareList -> at(currentSquare).getHouseCount() == 0 **and** self.squareList -> at(currentSquare).hasHotel())

6. Conclusion

μMonopoly is a software implementation of the famous board game Monopoly which provides a more limited set of rules but adds some other aspects to it in order to create a stronger bond with the OOP paradigm. The game is played by 2 to 4 users and the ultimate aim is making all the other players go bankrupt and thus win the game. The players start off with a certain budget and try to increase it to avoid going bankrupt easily. In order to do so they can buy or sell properties and build houses or hotels to increase the rent the opponents have to pay to him.

The way that we starting thinking for the design of this project started from making a detailed and carefully crafted list of functional and nonfunctional requirements. Those are the first frontier we faced when thinking about the realization of the object and they make possible the passage to designing the Use Case diagrams and made the descriptions of all the scenarios and cases in particular. The representation for these diagrams was aided by VisualParadigm software which models diagrams using the UML.

Afterwards, basing our thinking on the use cases and functional requirements, we starting building the object model for our project. The analysis started from the intuitive approach which would map all the application domain objects into Java classes and was followed by a more critical insight over the interaction and the core of the game logic which yielded the design of the solution domain objects as well. As soon as we had those scattered ideas of the object model, we went on connecting the ties and making a functional piece ready for implementation.

In order to understand the interaction of the objects in a particular scenario of the game, we built the Dynamic Models which are useful for us to have a clear insight of how our operation execution routine will be chained to the user interactions. We represented here the predicted flow of operations for the most important aspects of the game and the ones with an interesting behavior that are normally not easy to understand. A support for the dynamic behavior understanding was the State Chart diagram which crystallizes the procedure of operations and user interactions during a common scenario that is core of the game.

The following step was designing the system by extracting the design goals out of the non-functional requirements and organizing the classes into tight components to make up sub-systems with a unique function and whose classes have a strong bond. In this process we made decisions about the key concerns such as persistent data management, access control, security, global software control and finally, the boundary conditions. These were all part of the strategic thinking of the deployment of our system to a real one by maintain some principles described initially in the stages of creation.

In the final stage we worked on the Object Design and made an in-depth analysis of the implementation strategies by selecting the appropriate design patterns that would resolve some issues that would come up during the implementation. We chose among the Abstract Factory Pattern, Builder Pattern, Observer Pattern and Façade Pattern, also backed by Java Swing's Composite Pattern. These specified set of instructions with their resolutions provided us with a more regularly structured code and guaranteed proper functioning provided they are correctly implemented. In addition we used OCL to prepare some contracts about the most important,

significant and delicate methods in the classes. Ultimately, for every class we provided a detailed table of the class components: attributes and methods that make their interface in the respective subsystems.

Overall this is a process requiring a collaborative thinking process and carefulness to details and tricky twists of the design. Leaving a user with a lot of opportunities of how to interact with the system is sometimes a tradeoff made in order to ameliorate the user experience and increases the number of risky scenarios where the system might fail to do the proper thing. However, instead of channeling the user to one line of operations followed consequently, we have decided to take the deal and provide a better experience accompanied by a high quality software that will not be prone to failures and provide a real time game enthusiasm as if played with the real board.