CS 319

Object-Oriented Software Engineering Design Report

Backgammon

Group 1-D
Mert Armağan Sarı
Doğukan Altay
Berkalp Yılmaz
Ömer Sakarya

Contents

1. Introduction

1.1 Purpose of the system

Backgammon Game, all in all an implementation of traditional Backgammon game on the digital world. The game designed to achieve the old traditional feeling with the improvements of computers. The game offers, that two people can enjoy this traditional board game. Backgammon game consists of strategy and making move by predicting the future moves of your opponent, playing the game improves the decision making and making strategies of the player.

1.2 Design Goals

Before the composing the system it is crucial to identify the design goals of the system in order to clarify the qualities that our system should focus on. In this respect many of our design goals inherit from non-functional requirements of our system that are provided in analysis stage. Crucial design goals of our system are described below.

End User Criteria:

Ease of Use: The Backgammon Game ,as included in its name, is a game system. Our purpose is that, providing neat and useful design to the user in order to achieve ease of use in our System. The System consists of easy to use menus that users reach what they wanted to do in the System without lost. Also our System uses only Mouse to control. So this increases the ease of use in our System.

Ease of Learning: Although Backgammon is an old game, some young users or other users that had never seen the game before, can have troubles while playing the game. Since the game consist of bunch of rules. We offer users a " How to Play" menu, in order to teach users how the Backgammon game proceeds. We offer visual materials to achive the best learning experience for the user in order to increase the enjoyment from the game.

Maintenance Criteria:

Extendibility: All in all, for a software, keeping it updated and adding new features and components is crucial. So we designed the Backgammon game for future updates and features. Such as new rules, new play modes, etc.

Portability: Now the world is on the Mobile Stage. So portability of a software is crucial for its sake. In order the achieve this, we are determined that the system will be implemented in Java, since its JVM provides platform independency, our system will satisfy the portability.

Modifiability: In our system it would be easy to modify the existing functionalities of the system. In order to achieve this we will minimize the coupling of the subsystems as much as possible, to avoid great impacts on system components by a desired change.

Performance Criteria:

Response Time: Since our game based on turns, response time of the System not the first issue of it. However, a slow software always bothers the user, so we are trying to achieve the lowest response time in order to achieve best experience of game for our users.

Trade Offs:

Ease Of Use and Ease of Learning vs. Functionality:

In our system we determined that player should be able to learn and use the system very easily. Therefore our design proposes that the priority of the usability is higher than functionality. In other words our system does not bother the user with complex functionalities or we do not make the user to be lost in many functionalities, in order to make our system easy to understand and use.

Performance vs. Memory:

In the Backgammon, the System consist bunch of different objects. Such as, Board, Checkers, Dice, etc. To achieve the best performance, we need to manage the usage of memory very well. Otherwise we overwhelm the memory and game performance will be dropped and in some cases game could crash.

1.3 Definitions, acronyms, and abbreviations

Abbreviations:

MVC:	[2] Model View Controller
JDK:	[1] Java Development Kit
JVM:	[1] Java Virtual Machine

1.4. References

[1] http://en.wikipedia.org/wiki/Java_(programming_language)

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

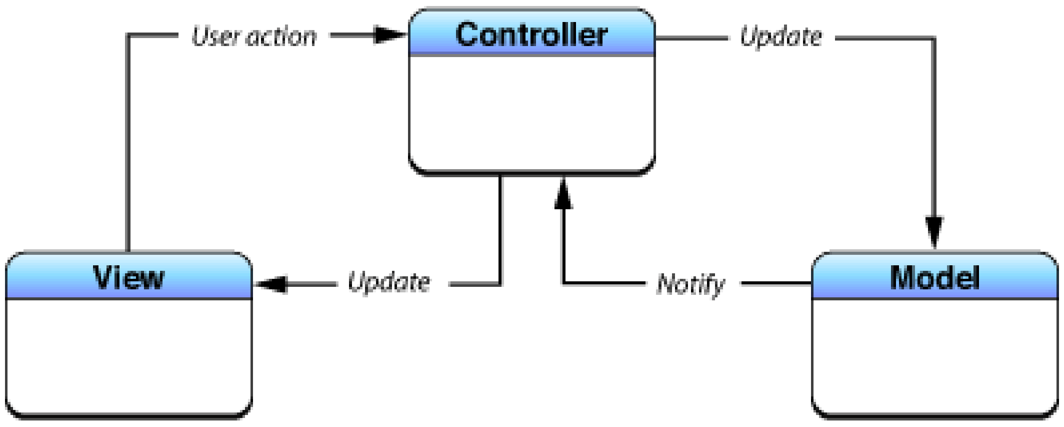
1.5. Overview

In this section, we represented purpose of the system, which is basically entertaining the player as much as possible, to achieve this purpose we defined our design goals in this part. Our design goals are determined according to provide the portability, ease of use, ease of learning, high performance, high maintainability.

2. SOFTWARE ARCHITECTURE

2.1 Overview

This section describes how our software architecture is. We are making a backgammon game which basically has a UI, takes inputs and puts out outputs, and stores the respective entities inside organised classes. Software world had encountered this problem before and developed architectures styles. We thought that MVC would

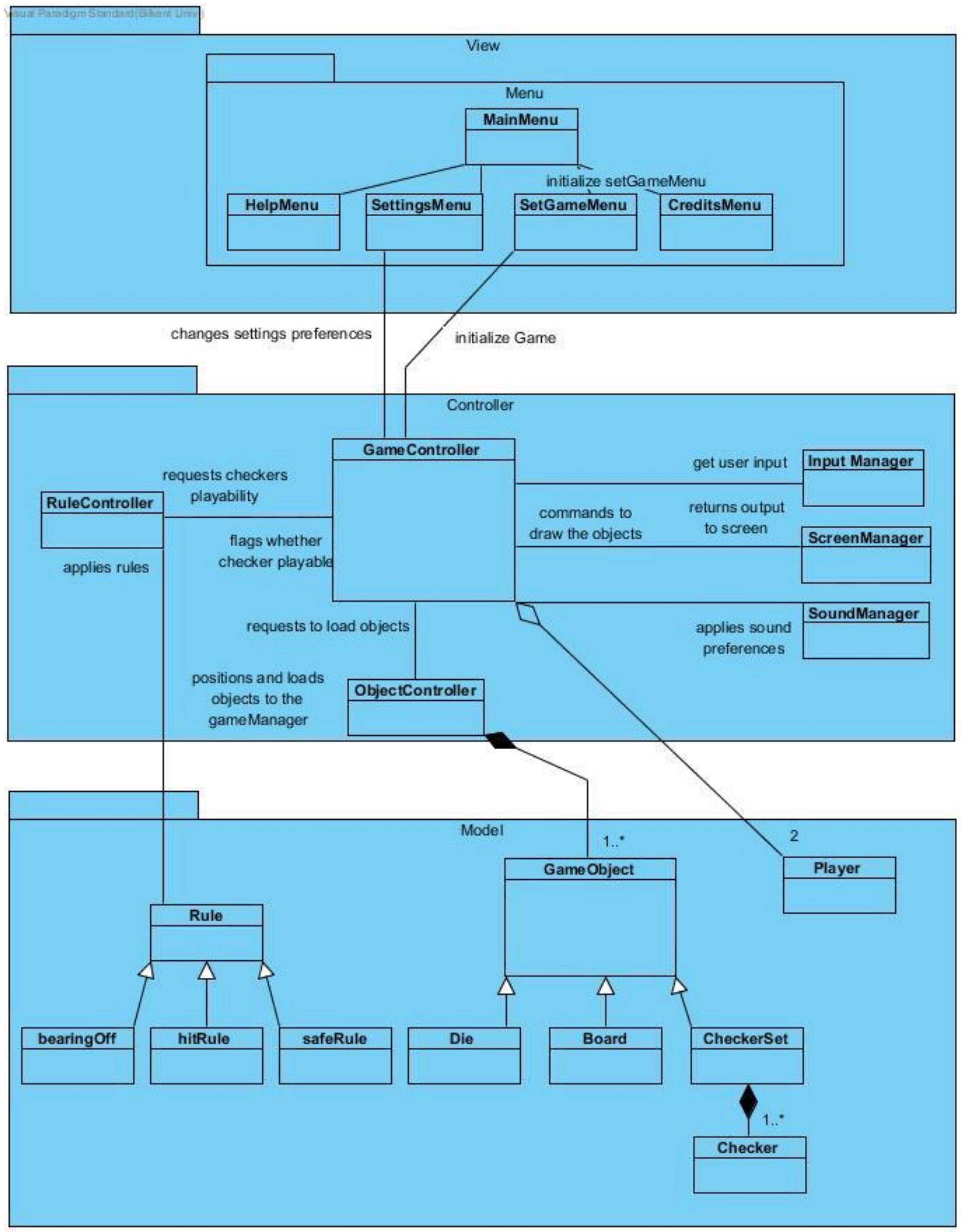


suit best for our solution. Below, is a very simple diagram of our architectural style:

Reasons why we though MVC would suit best are explained in 2.2: Subsystem Decomposition. Note that in the diagram above, associations between subsystems are not hierarchical. In essence, controller works as a intermediary between model and view subsystems.

2.2 Subsystem Decomposition

As mentioned earlier, our system is decomposed with respect to MVC architectural style. Below, is a figure showing details of the decomposition. We sticked to the MVC because our system is divisible as model, view and controller. We could have picked a different architectural style. However, considering good design includes high coherence and low coupling, this can be achieved through putting together our entity classes, manager classes, and UI classes separately. We have tried to put them together and saw that classes in each subsystem runs almost same operations. For instance, RuleController and ObjectController classes, which are in Controller subsystem, modify Rule and GameObject, respectively. This supports the idea of high coherence. Besides that, if we have sticked with MVC style, we can change our model classes (in other words, entity classes that hold information about in-game properties) and that change would obviously effect controller classes. In other words, controller classes' update methods have to change too. We think that this change is acceptable, because it also allows extensive changes to entity classes as well. Obviously, one can not change the whole game, converting the game from backgammon to chess, but can, for instance, change the implementation of rules to some other format. After that, only respective controller classes about the new format should be changed. No other changes would be necessary unless exceptions. Other architectural styles would cause way more complications than MVC.


Visual Paradigm Standard(Bilkent Univ)
View
Menu
MainMenu
HelpMenu
SettingsMenu
SetGameMenu
CreditsMenu
initialize setGameMenu
changes settings preferences
initialize Game
Controller
GameController
RuleController
Input Manager
ScreenManager
SoundManager
ObjectController
requests checkers playability
flags whether checker playable
applies rules
get user input
commands to draw the objects
returns output to screen
applies sound preferences
requests to load objects
positions and loads objects to the gameManager
Model
GameObject
Player
Rule
bearingOff
hitRule
safeRule
Die
Board
CheckerSet
Checker
1..*
2
1..*

2.3 Hardware & Software Mapping

Our system will be coded in Java. And since our game will be very simple, system requirements will also be modest. Hence, any computer that has a decent system properties and an installed java runtime environment can run our program.

Users are going to give inputs using a mouse and keyboard. Trackpad works the same way mouse does, in case the program gets ran on a laptop. Users will see an updated window from the computer’s window. So, that means any decent computer will be able to run our program.

2.4 Persistent Data Management

High scores will be saved as XML file. Besides that, sound effect are saved as their individual formats.

2.5 Access Control & Security

Our system is an off-line software. Users will not enter any authentication information, nor they will be entering critical information to the system. Hence, it’s security and access control is self-accomplished.

2.6 Boundary Conditions

Initialisation

- JRE is required on the system for our program to initialise and arrive at a playable state.
  - If JRE exists, when user opens the application, program starts to execute from where the program is in the disk.

Termination

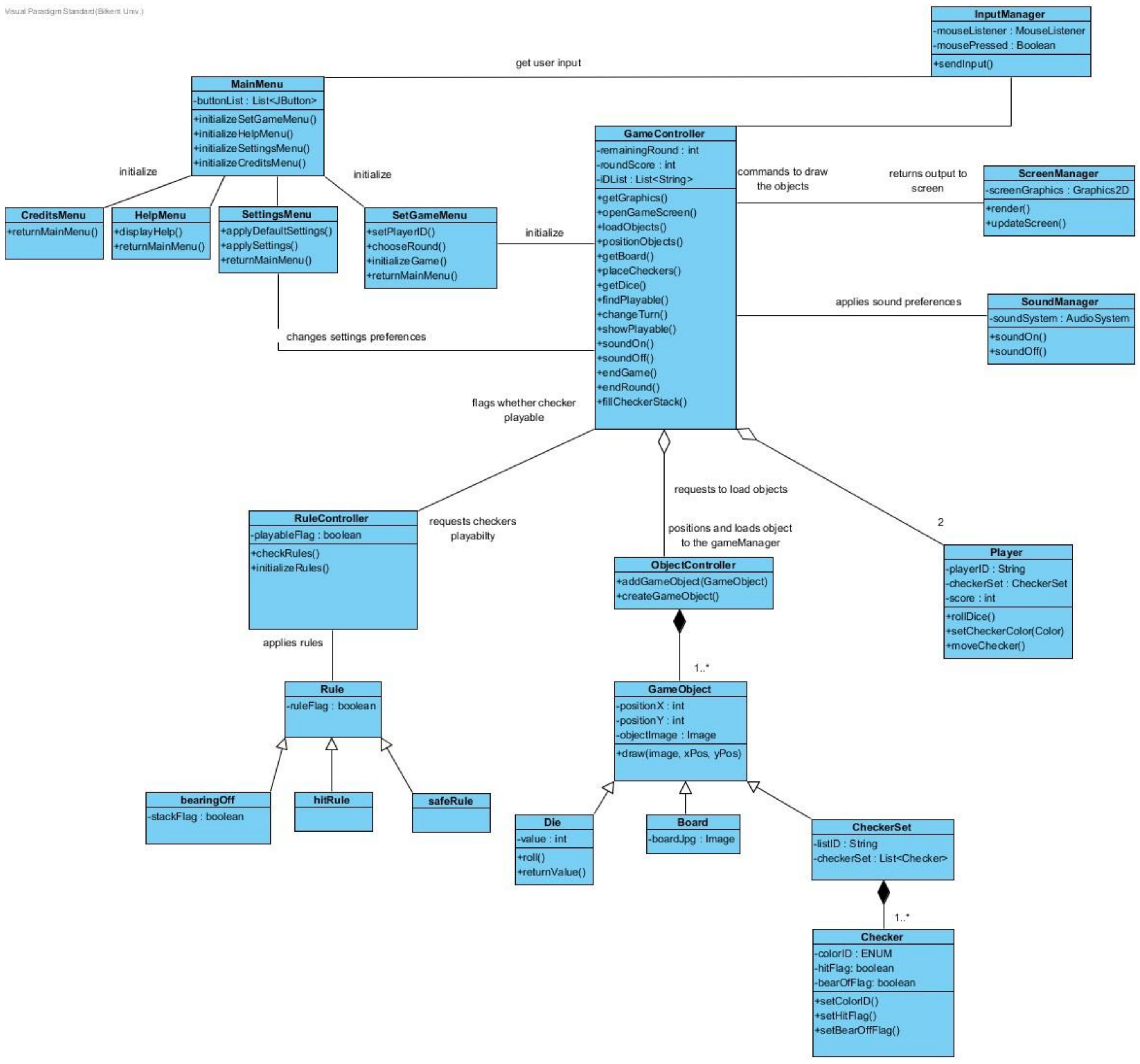
- User can quit the game at any moment by selecting quit from the menu. Although game process will be lost.

Error

- An unexpected in-game termination of the program will cause a data loss. In other words, game progress will be lost.

3. Subsystem Services

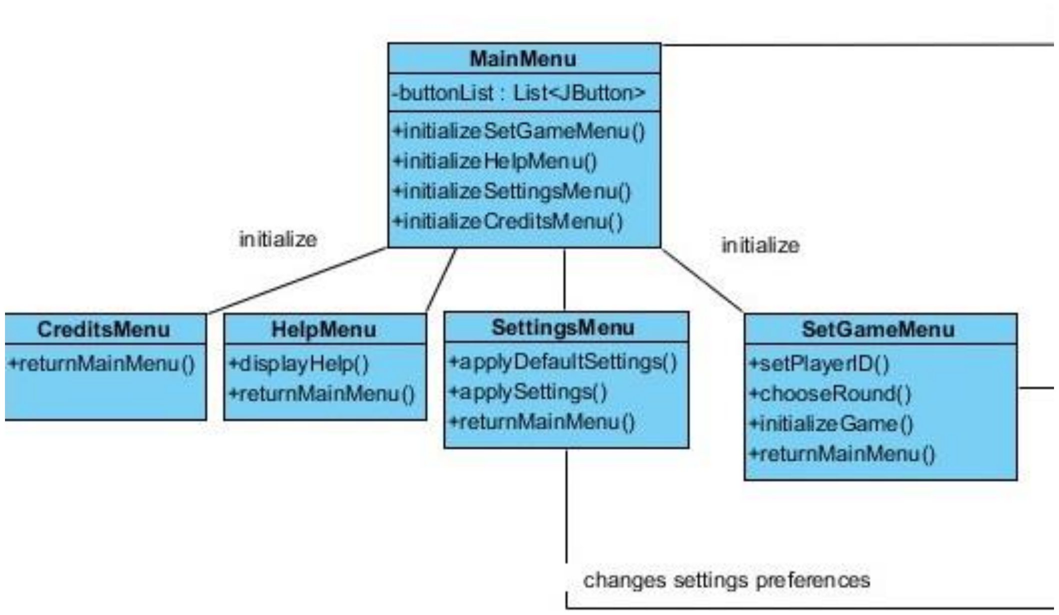
Class Diagram



3.1 View Subsystem

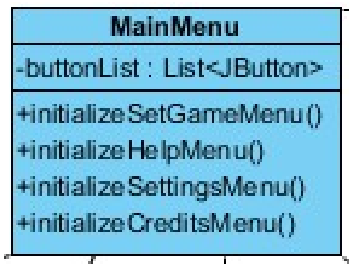
The classes included in this subsystem are basically to handle the graphical user interface of the program.

3.1.1 Game View



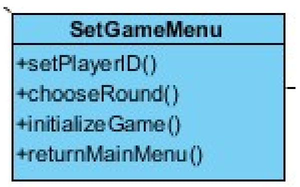
This subsystem handles the relation of the software with the user. The subsystem consist of interface related classes of the software . The classes can be considered as the GUI classes and it is responsible with understanding the mouse actions of the user in the system. Then sending the corresponding datas to the controller subsystem in order to update and control the game accordingly.

3.1.1.1 Main Menu Class



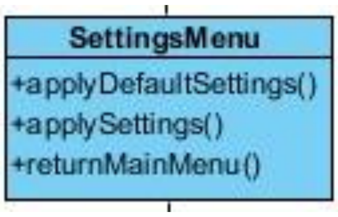
Main Menu screen is the opening screen of the game. It includes navigation buttons for opening other menus for the user. The buttons are Create-a-Game, Settings, How-to-Play, Credits buttons.

3.1.1.2 Set Game Menu Class



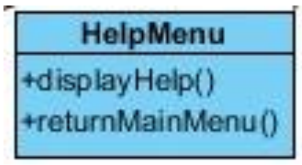
This window is the game creation screen of the game. User's can decide how many rounds they will play, their player IDs and start the game. This Menu consists of two text fields to fiil with IDs and 2 buttons to decide round count and a button to start the game.

3.1.1.3 Settings Menu Class



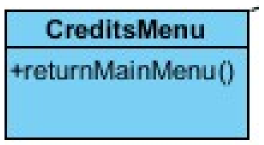
This menu is a screen that can user change the game attributes such as, sound on/off checker colors. The menu consists of a button for sound choice, 3 buttons for color choice of the checkers and a button for returning back to the Main Menu.

3.1.1.4 Help Menu Class



This menu is very straightforward. The menu is self has no buttons except the return Main Menu button and show the game rules as texts and some instructing images on the screen.

3.1.1.5 Credits Menu Class



This menu shows information about the developers of the game and some information about the system itself.

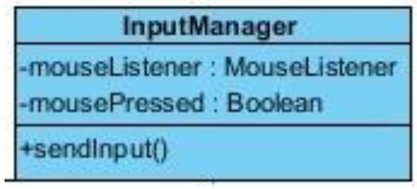
3.2 Controller Subsystem

The controller subsystem is for controlling the interaction between the player and the interface of the system. This subsystem controls the game objects according to the inputs gathered from View Subsystem. It is used for changing the game entities, rounds, checker colors and sound settings relatively to the inputs.

3.2.1 Input Manager

Input manager is used for getting the user's inputs to the system and sends those inputs to the GameController. The Input Manager is responsible for the user's interaction with the View Subsystem. GameController is strongly dependant to the Input Manager.

3.2.1.1 InputManager Class

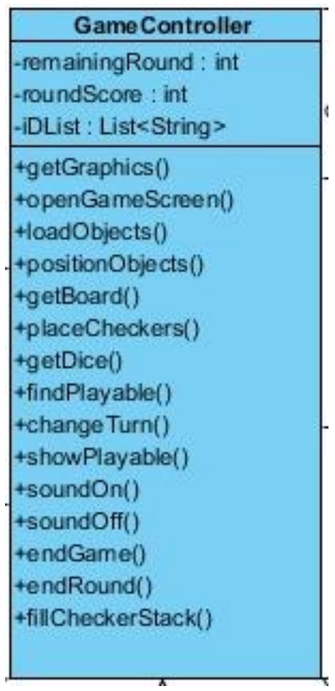


This class contains 2 attributes one of them holds Boolean and the other one is a MouseListener to retrieve the input from the user. There is one method called sendInput() which transmits the taken inputs to the MainMenu and GameController classes.

3.2.2 Game Manager

Game Manager holds the most important action in our system. It is basically the main controller part of the Controller Subsystem.Game Manager has lots of operations such as changing the sound setting by using the Sound Manager and so on. It also updates the checkers' movability with their places and players' conditions on the board. It is connected with both View and Model Subsystems to get input and calculate them within the game loop.

3.2.2.1 GameController Class

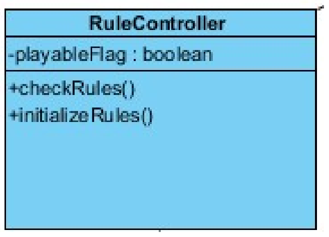


The GameController class is the most functional class in the Controller Subsystem. It handles most of the operations with other Managers and View Subsystem. It has 3 attributes and one of them is a List of Strings and the others are integer: remainingRound is for keeping the track of the round number so that the game will be finished if the round number is reached, roundScore is used for keeping the score of players to decide who will win the game, IDList holds the ID's of players and controls the other objects according to players. This class has many operations; **getDice()** is simply getting the Dice values for the players to make their move, **openGameScreen()** interacts with Screen Manager and opens the game screen, **getGraphics()** is simply collecting other graphical objects such as checkers and dice, **soundOn()** and **soundOff()** are used for controlling the sound of the game with the Sound Manager, **findPlayable()** and **placeCheckers()** are used for controlling the Checker objects with the Rule Manager, **endGame()** and is simply ending the game if the rounds are finished, **endRound()** is simply ending the round if one of the players' put all of his checkers to the stack, **fillCheckerStack()** is used for filling the checkers to the Stack in case of the players are at the bearing off stage of the game, **loadObjects()** and **placeObjects()** are used for getting the objects from the Model Subsystem and placing them according to their positions, **getBoard()** is used for getting the updated board from the Model Subsystem, **changeTurn()** decides which player is moving at that time in the game.

3.2.3 Rule Manager

Rule Manager is needed because of some rules in the game should be controlled uring the players are playing. This Manager is responsible for the keeping the track of the game's rules.

3.2.3.1 RuleController Class

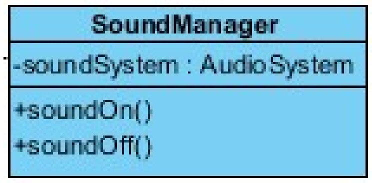


RuleController class is the main class of the Rule Manager and calls other works with the Rules class which is in the Model Subsystem. The class had one attribute of type Boolean to show the playable checkers according to Player's dice values. The class has checkRules() method to get information from the Rules class to determine if the Player's movement is valid and initializeRules() is simply initializing all the rules to check them.

3.2.4 Sound Manager

This Manager is used for playing sounds according to the user's inputs and the game phase. It interacts with the Game Manager to change the sound settings and play the suitable sound for the game.

3.2.4.1 SoundManager Class



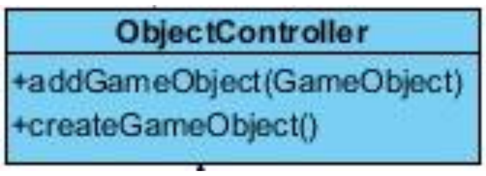
The SoundManager class has one attribute of type AudioSystem which is used in Java particularly. AudioSystem is used for playing the sound for the game. The soundOn() operation opens the sound in the game system to play every sound, the soundOff() operation closes the sound in the game system and these operations are connected to the

Game Manager to change sound settings.

3.2.4 Object Manager

Object Manager is controlling the game objects according to Game Manager. This manager is used for determining which game objects to create.

3.2.4.1 ObjectController Class

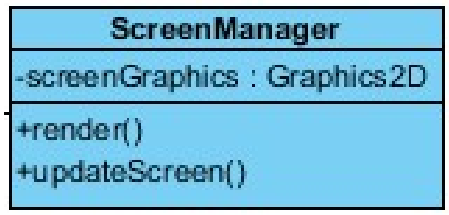


This class has no attributes but 2 operations. The createGameObject() method is responsible for the creation of GameObject and its children such as Die or Board. The addGameObject() method provides them to the Game Manager to show them in the screen with the Screen Manager.

3.2.5 Screen Manager

The Screen Manager is used for displaying the images of the objects and menus according to GameController class.

3.2.5.1 ScreenManager Class

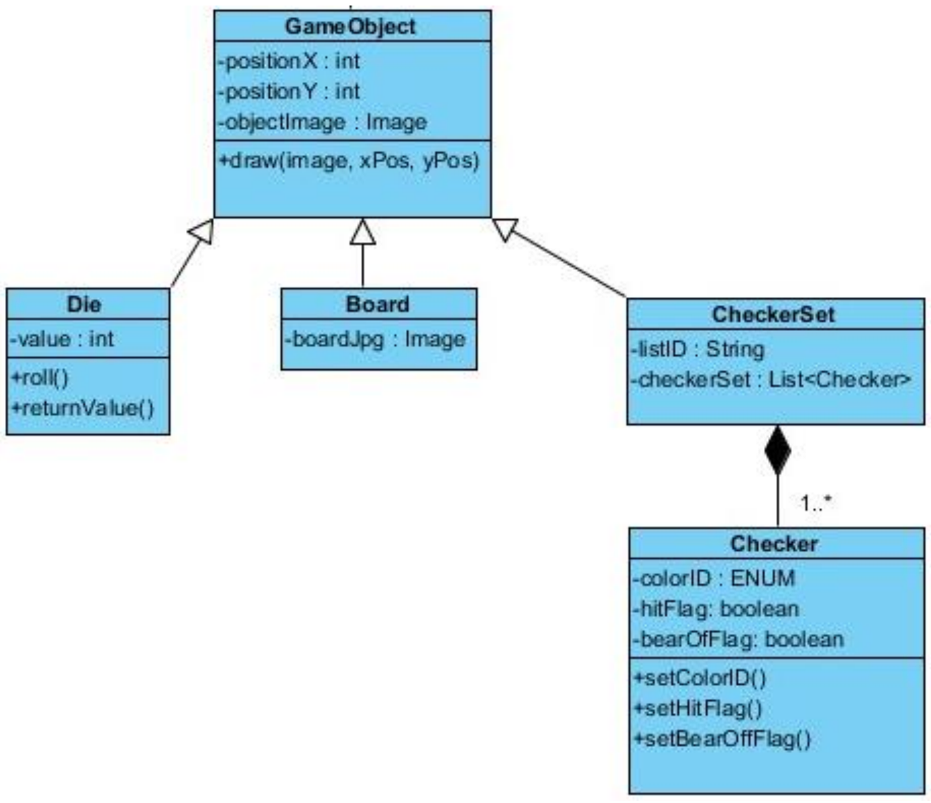


The ScreenManager class has one attribute which is a Graphics2D type and it is used for displaying screens and objects. There are 2 operations in the class; render() method is used for rendering the screen, pdateScreen() is used for updating the game objects on the screen to show the current state to the players.

3.3 Model Subsystem

The Model Subsystem's goal is that managing the game entities with the datas that came from controller subsystem and editing and updating the informations for the game entities and the game itself and provide the corresponding datas to the controller subsystem. The subsystem process the input datas for the game in order to follow the game actions and keeps the track of game entities.

3.3.1 Game Object Class



This class is the parent class for the game entities objects. It holds the information about the positions of the entities in the game screens. When game manager built the object controller, it uses this subclass to access the game entity values.

3.3.1.1 Die Class

This class is the entity class of the dice in the game. Contains the value of the dice as an integer and has two methods to generate and return the value of the dice.

3.3.1.2 Board Class

This class is the entity class of the board of the game. Contains an image of the board layout as an attribute.

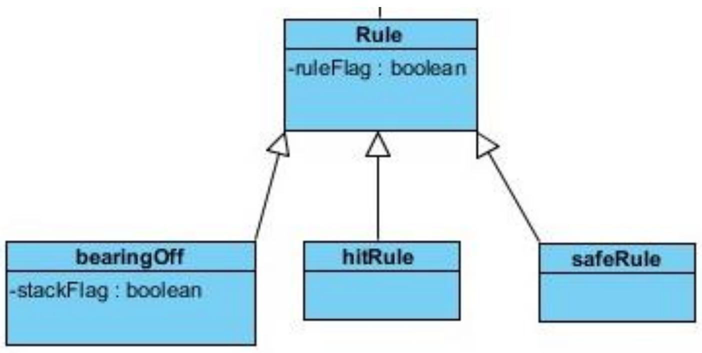
3.3.1.3 Checker Set Class

This class is the entity class of the checker set of the players. Consists of Checker objects. The object itself has two attributes as Id of the set and a data structure to contain the Checker objects in it.

3.3.1.4 Checker Class

This class is the entity class for the checkers in the game. It has three attributes as colorID as Enumerated type, hit flag and bearing-off flag as booleans to determine the stage of the checker.

3.3.2 Rule Class



This subclass is an entity parent class for the rules of the game. The Rule controller class in the controller subsystem creates this class after its initilization and provides data whether the move is playable or not and sends the information to the controller subsystem.

3.3.2.1 Bearing-Off Class

It is the class that contains the information and logic about the bearing-off rule. It determines whether the player is in the bearing-off stage or not. It has a boolean attribute to store the information about it.

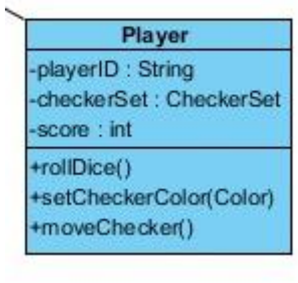
3.3.2.2 Hit Rule Class

This class contains the information and logic of the hitting checker rules as an entity class.

3.3.2.3 Safe Rule Class

This class contains the information and logic of the safe checker rules as an entity class.

3.3.3 Player Class



This class is an entity class of the players. Class consists of three attributions as player IDs , checker sets and score of the player. On the other hand, it has three operations as Rolling the dice, setting its checker set’s color and moving a checker.