Bilkent University
Department of Computer Engineering

Object Oriented Software Engineering Project Report

Project short-name: Curve Fever

Design Report

Zeynep Delal Mutlu, Yunus Ölez, Barış Poyraz

Course Instructor: Bora Güngören

Design Report
November 28, 2016

Table of Contents

1. Introduction

1.1. Purpose of the System

Curve Fever is the replica of the game “Curve Fever 2”. Our purpose is to create a game such that everyone playing from all age groups can enjoy and have entertainment. As the complexity of the system increases, it leads to complex implementations which will have a bad effect. As an example to that, performance can be given. Since our game will contain dynamic movements and it will handle event driven situations and the gameplay itself is dynamic it may have a possibility to create synchronization and performance problems. Therefore, the system that will be designed by us, will not be hard nor easy.

1.2. Design Goals

1.2.1. Reliability

We want our game and system to be reliable. It should not crash or give an error in runtime. Therefore, the system should be well designed.

1.2.2. Modifiability

The system we will design should be modifiable. This means that, the design must allow for modifying each component without modifying the other components. By using the object oriented design, we are planning to achieve this goal.

1.2.3. User Friendliness

The game should be easy to adapt and should not provide any conflicts. To achieve this, we should provide user friendly structure. Since we are wanting this game to be enjoyable for all the age groups, the logic of the game should not be complicated.

1.2.4. Good Documentation

The system has to be well documented for the developers. In order to achieve this, the reports should be clearly written.

1.2.5. Response time

The response time is one of the most important aspects of this project. Since our project is an interactive game, which consists of multiple players playing at the same time from the keyboard, there should be no delay in the game to provide better gameplay. For example, the system must be designed in such a way that, when the input is taken from the keyboard, it should immediately affect the movement of the player and update the view accordingly.

1.2.6.Readability

We aim to achieve writing readable code since writing readable code will allow us, the developers, to understand the source code and make changes accordingly.

1.2.7. Adaptability

Curve Fever should be playable in different environments. We will develop this system using Java. Thus, users having a Java Runtime Environment can play the game.

2. Software Architecture

2.1. Subsystem Decomposition

To be able to achieve our goals, system decomposition, which will discussed in details within the name of subsystem decomposition, should follow the design and provide all our classes in a proper, organized way. Subsystem allows us to collect all classes that have associations, events and allows us to relate with each other. Therefore, classes with similar functionalities are put in the same subsystem components. This is done with respect to Model - View - Controller architectural style.

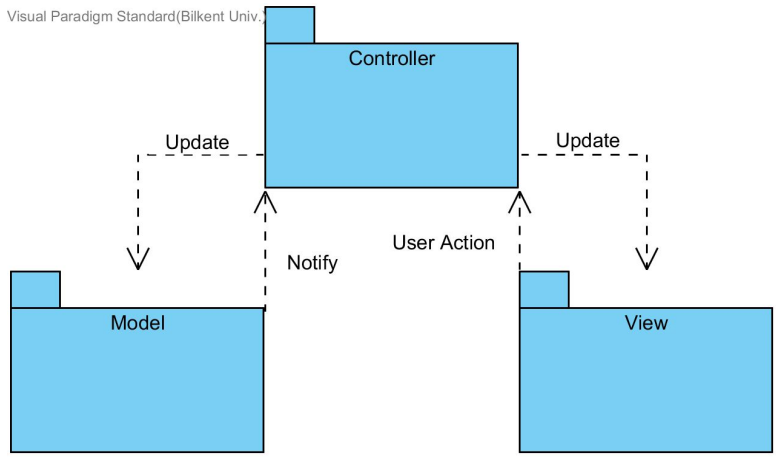
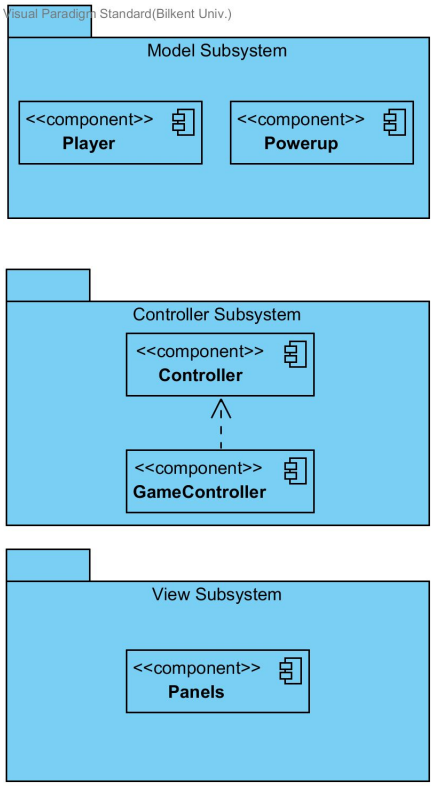


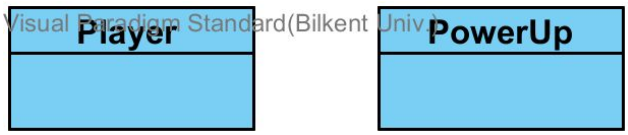
Figure 1. System Decomposition

Model - View - Controller Architectural Style is shown.

In the figure above, MVC architectural style is shown. As the name suggests, MVC divides software application into three parts: Model, View & Controller. These parts have some relations with each other. View contains any output which can be seen by the user and it may allow to get some inputs, which can be said user actions. The user action is taken by the controller which accepts the input and transmits what happened to both model and view. Model basically represents an object.



2.1.1. Model Subsystem



The first subsystem of the program is the Model unit. The main purpose of this system is to have representations of model objects and relationship between them. Within the model subsystem there are 2 classes. These classes are Player and PowerUp classes. These classes have must have relations with some of the controller classes. Powerup class contains the informations of powerups, which in our game changes the gameplay, resulting in an event-driven game. Player class contains the informations of players and has a relation with both GameController and MoveController class. GameController class contains the positions of the players and MoveController class will contain the information of key configurations.

2.1.2. Controller Subsystem

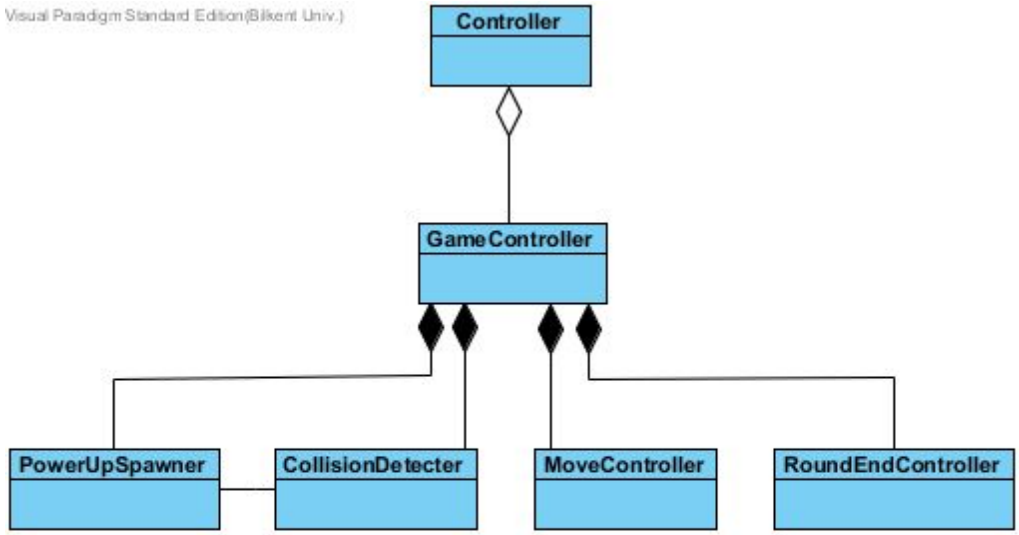


Figure X. Controller Subsystem
This figure consists of 6 classes which are the part of the controller subsystem.

The second subsystem of the program is the Controller unit. The main purpose of this subsystem is to direct the hole system. It accepts inputs from the other subsystems and commands them instructions. It basically consists of 6 different classes with different hierarchies. There are 4 classes, which are linked to the GameController, called PowerUpSpawner, CollisionDetector, MoveController and RoundEndController. These 4 classes handle some cases and inform the GameController in necessary situations. These situations consist of checking whether the round has ended or not, getting inputs from the keyboard, detecting and determining collisions and spawning power-ups at random times and locations. However, these classes only help the GameController class in some cases and overall, the rules of the game are determined and the game is controlled by the GameControler. This class also asks the View Subsystem to create and modify view elements when needed in the game. Other than that, there is a class called the Controller which is at the top of the hierarchic order. This static class is the manager of the whole program and it also communicate with the View Subsystem in order to create new scenes.

2.1.3. View Subsystem

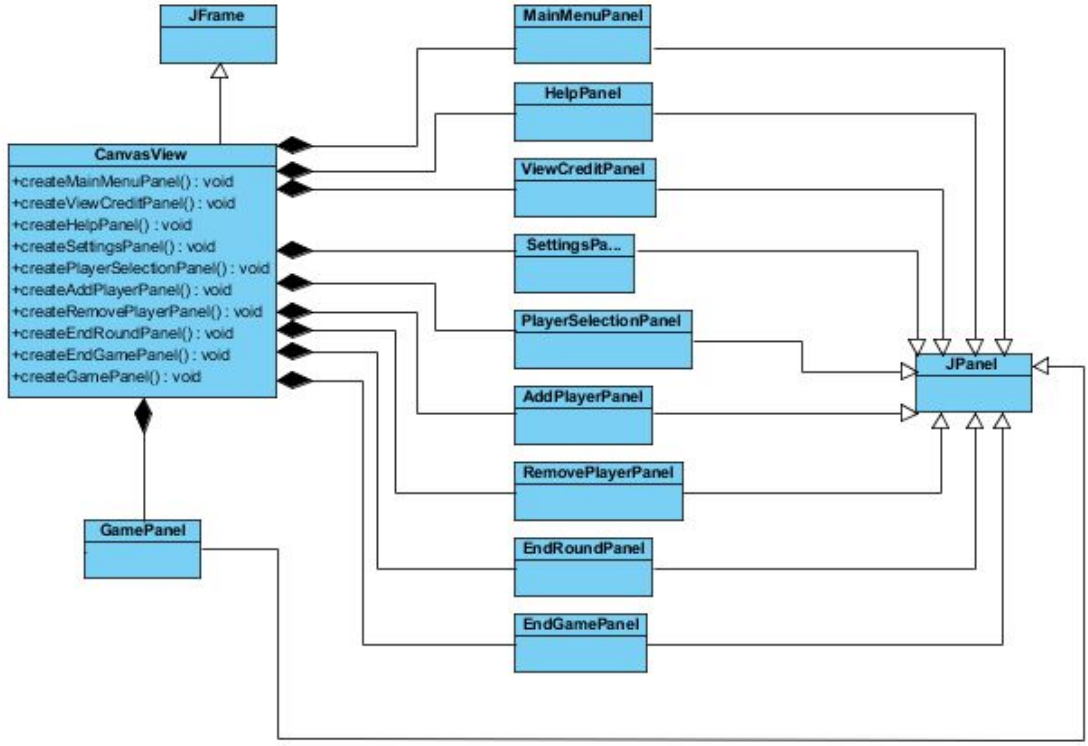


Figure X. View Subsystem
This figure consists of 13 classes which are the part of the view subsystem.

Our third and the last subsystem is "View Subsystem". In this subsystem there are 13 classes and the main goal is that creating an interface between play and players. The determinant class here is "CanvasView" class. When any panel is needed in Curve Fever, "CanvasView" class basically gets in contact with this specific panel class and the panel is created. "CanvasView" class works with JFrame. Therefore, after a panel is created, it is embedded into frame.

There are 10 classes which creates panels. Each panel class should use the JPanel class which is used for visual programming. These ten classes' names are easily understanble, even so they should be indicated in detailed as follows:

MainMenuPanel: Creates main menu which includes buttons for adding player, settings and help. When the main menu is needed, "CanvasView" class runs the function 'createMainMenuPanel()'. Therefore, main menu panel is created.

HelpPanel:Creates help menu which includes 'How to Play' instructions. When the help menu button is clicked in the main menu, "CanvasView" class runs the function 'createHelpPanel()'. Therefore, help panel is created.

SettingsPanel:Creates settings menu which includes certain setting options which will be decided later. When the settings button is clicked in the main menu, "CanvasView"

class runs the function 'createSettingsPanel()'. Therefore, settings panel is created.

ViewCreditPanel: Creates credit menu which includes following sentences:
"This project is part of CS319 course in Bilkent University" and also designers name and also special thanks to our instructor. When the credit menu button is clicked in the main menu, "CanvasView" class runs the function 'createViewCreditPanel()'. Therefore, credit panel is created.

PlayerSelectionPanel: Creates player selection panel which asks the number of players. When the player selection button is clicked in the main menu, "CanvasView" class runs the function 'createPlayerSelectionPanel()'. Therefore, player selection panel is created.

AddPlayerPanel: Creates adding player panel which asks for each player's name, key configuration, and color. When the player selection button is clicked in the main menu, "CanvasView" class runs the function 'createPlayerSelectionPanel()'. Therefore, player selection panel is created.
We are planning to add a new panel class which is called "PlayerScreen". This is called automatically by "CanvasView" after the adding process is finished. In this panel, we can see the all players' information, play button and remove button. If player clicks play button, "CanvasView" understands that game will start. If player clicks remove player button, "CanvasView" understands that “RemovePlayerPanel” class will be used as a next step.

RemovePlayerPanel: Creates remove player panel which asks for a player which is wanted to be deleted. When the remove player button is clicked in the "PlayerScreen" panel, "CanvasView" class runs the function 'createRemovePlayerPanel()'. Therefore, remove player panel is created.

EndRoundPanel: Creates the last discontinued round's panel which shows that each player's score in that round. When the round is finished, "CanvasView" class runs the function 'createEndRoundPanel()' which shows the scores each player gained. Therefore, end round panel is created.

EndGamePanel: Creates the completed game panel which shows that each player's overall score in the game. When the game is finished, "CanvasView" class runs the function 'createEndGamePanel()' which shows the scores for each player Therefore, end round panel is created.

GamePanel: is an active playing field. This class works with “GameController” class simultaneously. “GameController” class informs “GamePanel” class for each player’s direction and the game panel will be changed by these inputs. Why do not we use Controller to indicate the direction to “GamePanel”? There are some reasons. First one is about collision detection mechanism. In order to catch collision, “GameController” class checks the coordinate of each player’s head pixel by pixel. That means, “GameController” should take information directly from “GamePanel”. The other reason is about the speed. There are too many instantaneous inputs and

outputs between “GameController” and “GamePanel”. If we use “Controller” for each input and output, it makes our game tired.

2.2. Hardware/Software Mapping

Curve Fever will be implemented using Java’s latest version. Therefore, the computer which the game will be runned should support Java, it’s latest version, and have an operation system. Also, in order to play the game a monitor, a mouse and a keyboard will be needed. The monitor will be used to display the game and it’s contents, the mouse will be used to press GUI elements such as buttons to select the color of the player and finally, the keyboard will be used to enter some strings such as names and to move the player with the determined key configurations. Also, in order to save data like high scores and player names into a text file, the system which the program will be runned should support .txt file format.

2.3. Persistent Data Management

At the end of each game the number of games won by the winner will be incremented by one and will be written in a file in order to display that data in the following games. In other words, the number of wins each player have will be saved as a persistent data. To add such a functionality to the program the Controller class should be able to read and write from the .txt files.

2.4. Access Control and Security

Curve Fever does not need Internet and any server connection in order to hold users' datas and to run the game. Addition to that, the game will be played on just one computer. That means there is no need any network protocol adaptation.Therefore, access control is not considered as a security problem.

On the other hand, there should be some control mechanisms to check users' datas. We are planning to hold each player's data in a .txt file(or any similar text file). In order to make this file logically unalterable, we are planning to use some methods. Firstly, the datas should be hashed and then salted. Hence, players' names and scores cannot be seen and changed by someone else. The other control mechanism is about checking name similarity, key similarity and color similarity among players. If either player chooses another player's name, color or keys; our controller will warn the player in order to change his/her information.

2.5. Boundary Conditions

2.5.1. Initialization

There will be an executable file to start the program. When the user opens this file, the system will be initialized. The Controller class have the first code to run and it will send a pulse to the View Subsystem to create and view the main menu. Also, the Controller class will read a text file to load high scores from old games.

2.5.2. Normal Termination

Users can exit the system using the exit buttons in the GUI or directly by closing the program's window. However, before closing the program the Controller class should handle saving high scores and player names. Therefore, when a user requests to exit the program the Controller should write the data, which includes high scores and names, to a text file to use them in later executions.

2.5.3. Midgame Termination

If one of the players decides to exit the game while playing using GUI buttons or directly by closing the program's window, the system should not do anything but exit. Since, the user requested to exit while playing the game, there is no need for saving new high score data.

3. Subsystem Services

As discussed earlier, system is decomposed into three subsystems which are Model, View, Controller.

3.1. Services of the Model Subsystem

Model classes which are the representations of objects compose the Model subsystem. It contains important knowledge, background information on objects and it provides necessary informations for the Controller subsystem. Then, Controller subsystem uses this data, gets the data from Model class to handle game operations. For example, in our game Player classes contains the key configurations of the respective player, thus allowing controller to check whether that input is taken or not. If it is taken, it can make corresponding changes.

3.2. Services of the Controller Subsystem

In this subsystem, as it can be seen from the earlier parts of our report, we grouped all of the controller objects to manage game operations. Controller subsystem is the most important part of our system and it can fill the gap between View and Model. When, user interacts the system within a panel, controller is responsible to take the action, handle what needs to be done, notify others and make changes.

3.3. Services of the View Subsystem

This subsystem consists of all panels that user interacts with. Since a user action leads to changes via done by controller, views, in other words panels, should also be changed. To satisfy this, we have set up each screen as a panel. These panels will be created and shown by CanvasView which will either update or create views according to the current state.

4. Low-Level Design

4.1. Object Design Trade-Offs

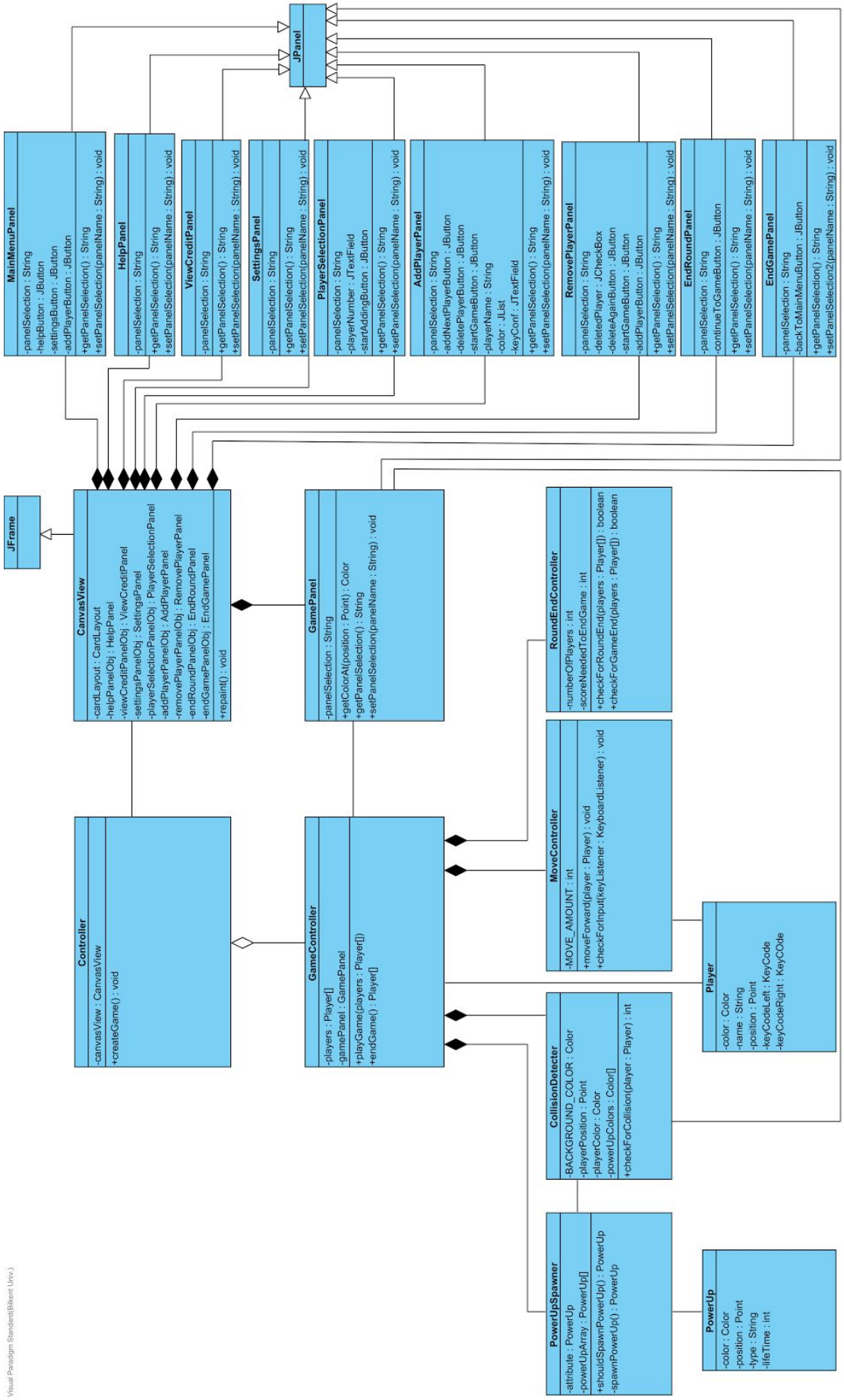
4.1.1. Space - Time / Time - Memory Trade-Off

Space refers to the storage consumed in performing tasks. Time refers to a response time, or a computation time meaning that how fast it have been done. In terms of our game, the most important part we can see this trade-off is in the gameplay itself. Since the gameplay is dynamic with real-time events occuring, event-driven game, and also since the game needs to get, set things and create needs fast, for that part of the game, we have to make a trade-off. Within the needs of our project space is not a big deal, but time is a concern, so we should try and find methods, try different algorithms and design our system better in order to catch up with dynamically occurring events.

4.1.2. Rapid Development vs Functionality

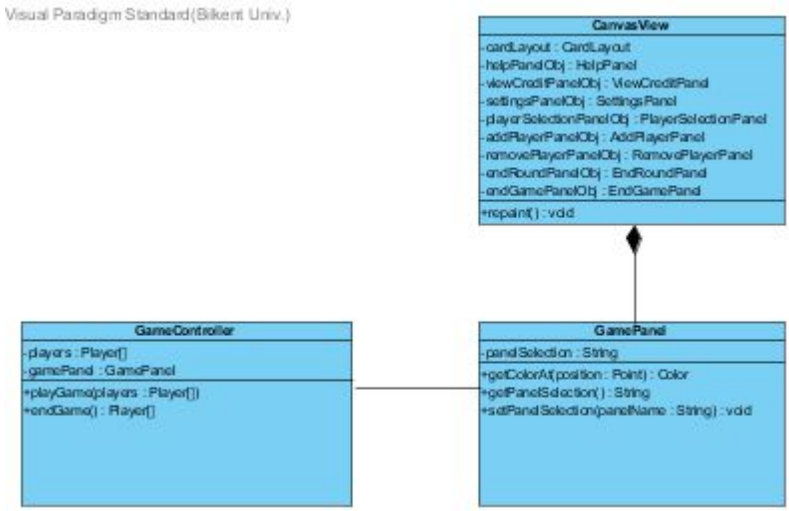
Since we have a limited time for the project, though this is not our expectation nor aim, we might have some trade offs in order to finish this project on time. But, we believe that, good usage of time will be the key for this project and we are aiming to finish with everything set. The reason this is given as trade-off because this is possible for any project.

4.2. Final Object Design



4.2.1. Façade Pattern

Between subsystems, a communication needs to be satisfied in order to have data and communication between model, view and controller. Façade pattern is used to reduce the complexity of grouping up the classes and subsystems that are connected. In our project, façade pattern can be done on our GameController class. Since GameController is the communication for both view and controller, it has been given properties to handle the classes down it. In our project Façade design pattern is as like this. Also CanvasView is using the façade pattern, since it provides all the transitions between different screens.



4.2.2. Singleton Pattern

In order to have only one instance of some classes at a time, the Singleton pattern is used in these classes. The Controller class, for example, uses this pattern. Since, this class is the main controller and since it handles most of the cases, there should only be one instance of it. Also, this instance should be reachable from other classes so, there should be a public function inside the Controller class which returns this Controller instance. Also, since the CanvasView class uses the façade pattern, it should also implement the Singleton pattern. This is because to ensure that each of the panel classes that reaches the CanvasView class, accesses the same instance of the CanvasView class.

4.3. Packages

In this part of the report, package diagrams are given according to the final class diagram that is given in section 4.2 Final Object Design.

4.3.1. Model Package

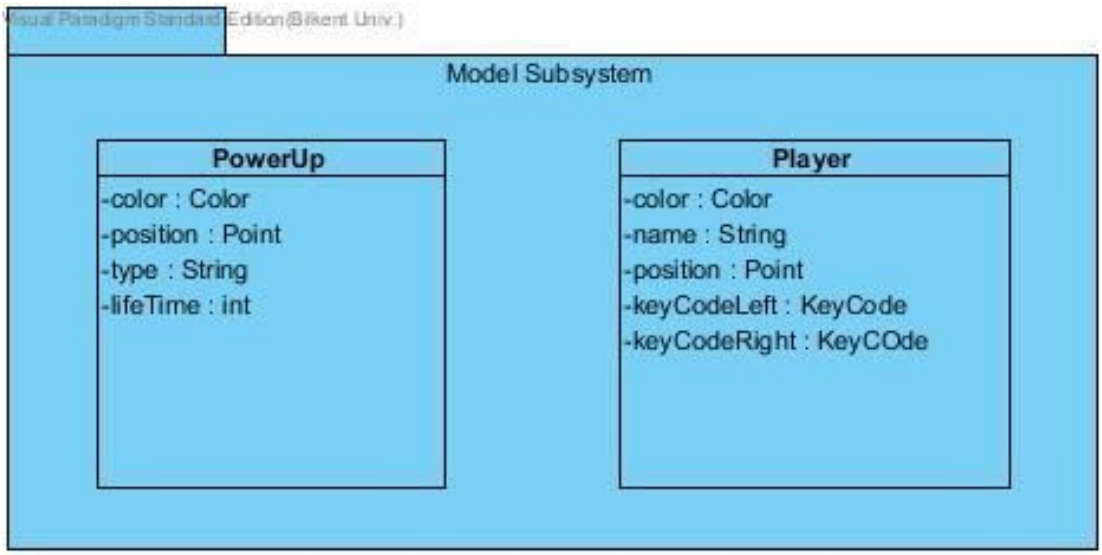


Figure 1 – Model Subsystem

In the model subsystem, there are two classes; Player and PowerUp. Both model classes have direct relationships with Controller Subsystem. When we get the information of players in the View Subsystem, player objects are automatically created and this process is controlled by GameController class which is the part of the Controller Subsystem. Powerup class is little bit from that. When the GameController object is created, PowerUpSpawner class produces some objects from PowerUpClass. We foresee that there is no direct connection between View Subsystem and PowerUp classes. We handle powerups by using PowerUpSpawner class which is part of Controller Subsystem as well.

To check collision, PowerUp class send its variables such as position and color to PowerUpSpawner class, so there will be some process between PowerUpSpawner and CollisionDetecter, but these are all about controller system's processes. Model classes do not have control mechanism. Model classes just provide information for Control Classes.

To sum up, models classes include information for controller classes. While Player class has a relations between both GameController and MoveController classes, PowerUp class has a relation between just PowerUpSpawner class.

4.3.2. View Package

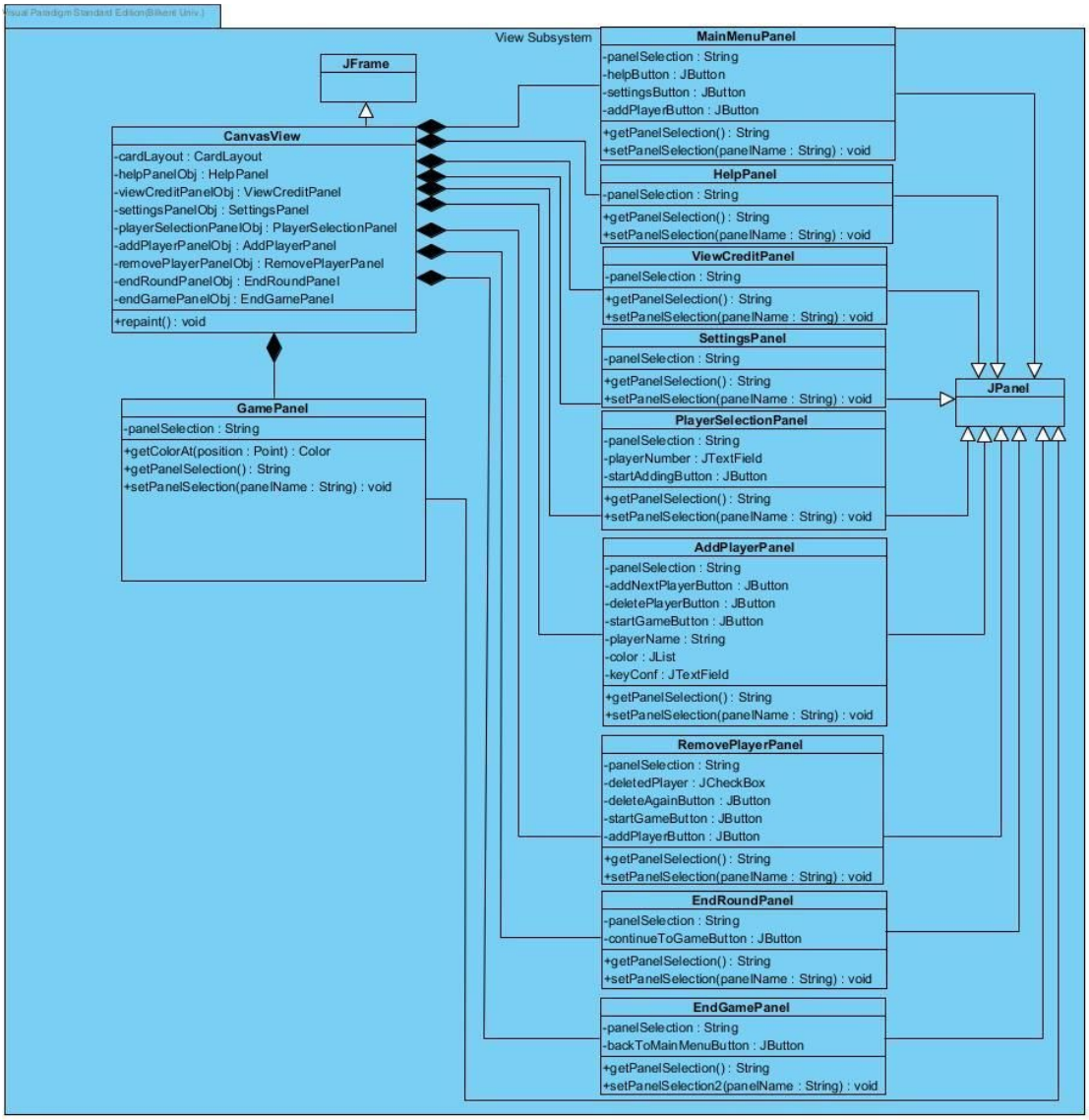


Figure 2 – View Subsystem

In "View Subsystem", we manage whole view classes such as panels and frame. We also get some important information from players such as key configuration information, each player's name and their colors. Therefore, it can be said that some functional operations are going to be handheld in addition to visual operations.

We foresee some important points in this subsystem such as timing and interpass view classes. As you can see in the figure, there are not any directly connection between panel classes. That means, we control all panels in the CanvasView class. For example; in the "MainMenuPanel", we will put 'Help' and 'Settings' buttons. Popularly, when we press Help button, it is supposed to create

HelpPanel by using ButtonListener in the MainMenuPanel class. However, our design is little bit different from that. We are planning to inform CanvasView class when we press Help button in the MainMenuPanel class. Hence, CanvasView class can be going to create HelpPanel object by itself. Under favor of this design, we are planning to minimize complexity of the system, because there will be huge amount of passing operations among classes.

In addition to that panel selection, we are planning to use CardLayout class. By using this class, it will be possible to choose required panel easily. It can be thought a kind of ArrayList which holds all panel objects.

As we indicated above, great majority of panel classes are going to be managed by CanvasView class. However, GamePanel class is an exception. It is going to be controlled by different understanding after first creation. We need some instantaneous variables from GamePanel class such as positions of players in order to check collision. For that purpose, it may be hard to get these information over Controller and CanvasView classes, because it is quite hard for implementation and it is time consuming. Therefore, we create a direct connection between GamePanel class and GameController class. Nevertheless, it is still a part of CanvasView class, because it is panel and it uses JFrame and Jpanel classes as well.

View Subsystem has connections with both Controller Subsystem and Model Subsystem. These connections will be stated in detail under the title of "Pattern" in the upcoming parts.

4.3.3. Controller Package

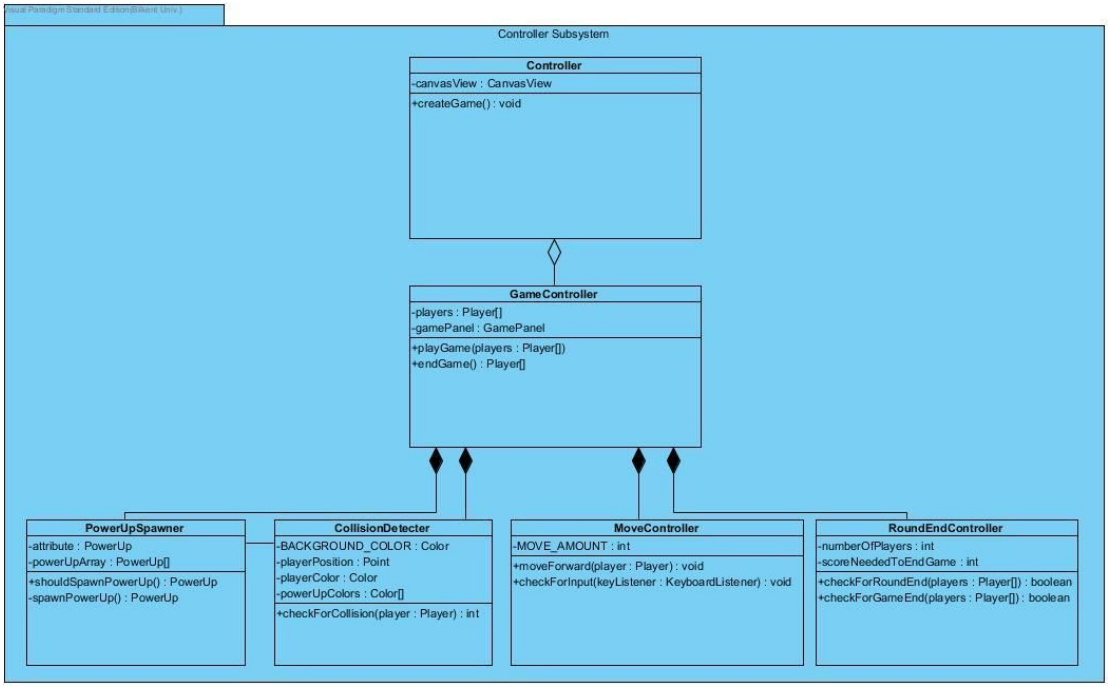


Figure 3 – Controller Subsystem

According to our design, Controller Subsystem controls the whole system and organizes transportation of information between both subsystems and classes.There are four controller classes which are PowerUpSpawner, CollisionDetecter, MovieController and RoundEndController.

PowerUpSpawner choose some locations randomly and and check whether there is collision or not. If there is no collision then it creates some PowerUp objects. CollisionDetecter works with both model classes Player and PowerUp. It is simultaneously check the GamePanel by using coordinate system. If there is collision, then it warns GameController class.

MoveController is all about the interaction between each player and detection of his/her pressed key. If he/she press right key, MovieController class will warn GameController. And GameController give this information to GamePanel to draw it. RoundEndController controls if the game or the round is finished or not. If the round is finished,this information goes through the way: GameController-->Controller-->CanvasView-->EndRoundPanel. Therefore, players can select what they want to do as a next step from this panel.

There is also two controller classes as well. One of them is GameController. As you can see in the figure, GameController class manages whole the game logic and game processes. It has to work with view class and some control classes simultaneously. This means there should be very elaborare work in this design. The other controller

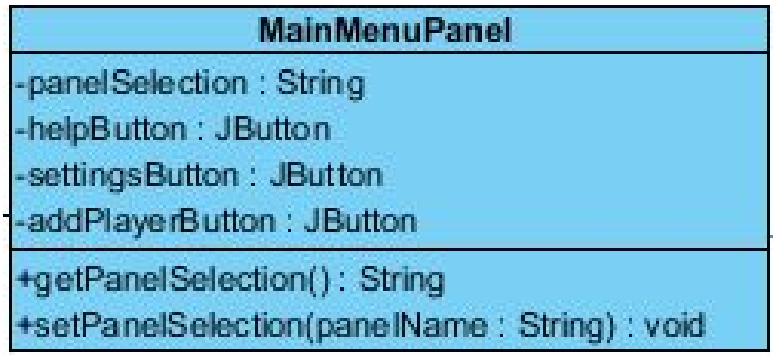
class is Controller class. Not only the game but also whole system is managed from this class. If there is no command to start the game from users, game will never start. However, when we open the system, Controller has to start. Both logic part of the system and view part of the system have to stay loyal to Controller class.

As we mentioned in the previous sections, Controller class gets the players informations from the CanvasView. Since CanvasView provides the screen and the places where players can type information or select from colors, the screen helps us to take input and transmit them to the GameController class. When rounds and the game finish, the each player's information will be updated in the Controller.

4.4 Class Interfaces

4.4.1 User Interface Classes

4.4.1.1. MainMenuPanel

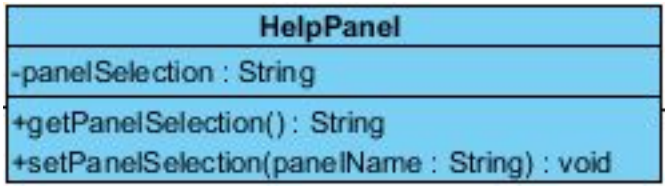


CanvasView creates a MainMenuPanel object which draws main menu panel and this panel includes buttons for adding player, settings, view credits and help. This main menu object is hold in the cardlayout object in the CanvasView class. Hence, when the main menu panel is needed, CanvasView class can reach this view easily.

MainMenuPanel class also has a String value which includes what is the pressed button information. There are also getting and setting methods for this String value. CanvasView has already created a MainMenuPanel object. Hence, CanvasView class can get this String value by using getting method. From that point, CanvasView class decides which panel object will be selected from the cardLayout based on the String value.

4.4.1.2. HelpPanel

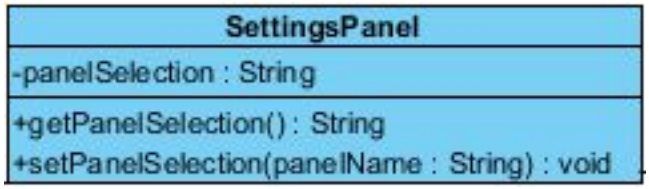
Creates help menu which includes 'How to Play' instructions.



When the help menu button is clicked in the main menu, String value becomes "help" and CanvasView class get this information by using getting method. Finally, HelpPanel object will be selected from the cardLayout and showed in the screen.

4.4.1.3. SettingsPanel

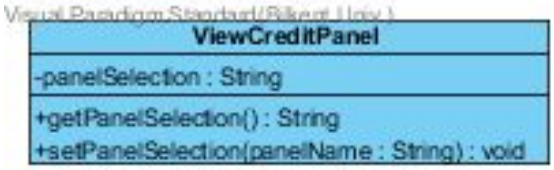
Creates settings menu which includes certain setting options which will be decided later.



When the settings menu button is clicked in the main menu, String value becomes "settings" and CanvasView class get this information by using getting method. Finally, SettingsPanel object will be selected from the cardLayout and showed in the screen.

4.4.1.4. ViewCreditsPanel

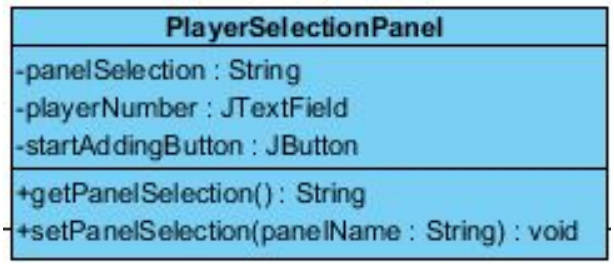
Creates credit menu which includes following sentences:
"This project is part of CS319 course in Bilkent University" and also designers name and also special thanks to our instructor.



When the ViewCreditPanel button is clicked in the main menu, String value becomes "viewcredits" and CanvasView class gets this information by using getting method. Finally, ViewCreditPanel object will be selected from the cardLayout and showed in the screen.

4.4.1.5. PlayerSelectionPanel

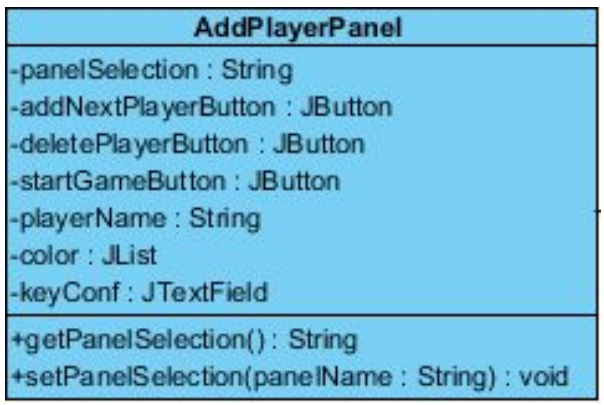
Creates player selection panel which asks the number of players.



In this class, there are extra GUI elements such as JtextField which gets the number of players and JButton which makes the String value "addPlayer". Hence, CanvasView class gets this information by using getting method and selects the AddPlayerPanel object from the cardLayout and showed in the screen.

4.4.1.6. AddPlayerPanel

Creates AddingPlayerPanel object which asks for each player's name, key configuration, and color.



When the addPlayer button is clicked in the PlayerSelectionPanel, String value becomes "addPlayer" and CanvasView class gets this information by using getting method. Hence, AddPlayerPanel object from the cardLayout will be selected.

In this screen, we see there different parts for each player; name, color and key configuration. Name and key configuration variables are taken from JtextField objects. In order to make a color choice, Jlist can be used. If we use Jlist, clicking is enough to choose color.

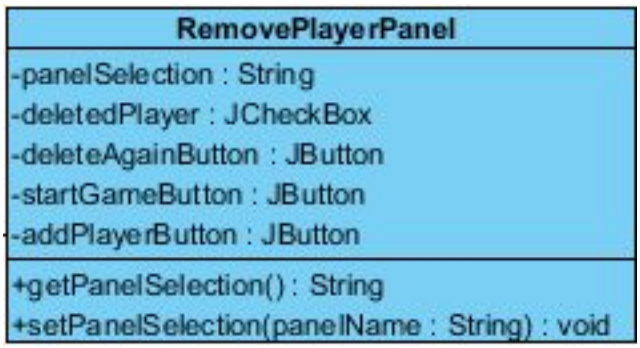
There are also three different Jbutton. These are used for adding another player, deleting player and starting game. Players press addNextPlayerButton in order to enter another player's information. Hence, String value becomes "addNextPlayer" and CanvasView class gets this information by using getting method. Therefore, another addPlayerPanel will be created for another player. When the adding is finished,

deletePlayerButton and startGameButton will be activated. If player presses deletePlayerButton, String value becomes "deletePlayer". Therefore, CanvasView class selects the RemovePlayerPanel object from the cardLayout and showed in the screen. If player presses startGameButton, String value becomes "startGame" and CanvasView class gets this information by using getting method. Therefore, CanvasView class selects the GamePanel object from the cardLayout and showed in the screen.

There is also a functional part about adding players. When the each player's information is ready, we are planning to send these information to Controller class, because according to our design Controller provides a connection between View Subsystem and Game Controller class. Even if there is no certainty about transmission of players information, it seems that it can be handled in CanvasView class. Some functions which take information from AddPlayerPanel will be written in CanvasView class, and then this function will be used in Controller class by using CanvasView class's object. Hence, Players array is created in Controller. GameController will take this array from the Controller class. Thus, the game can be started.

4.4.1.7. RemovePlayerPanel

Creates remove player panel which asks for a player which is wanted to be deleted.



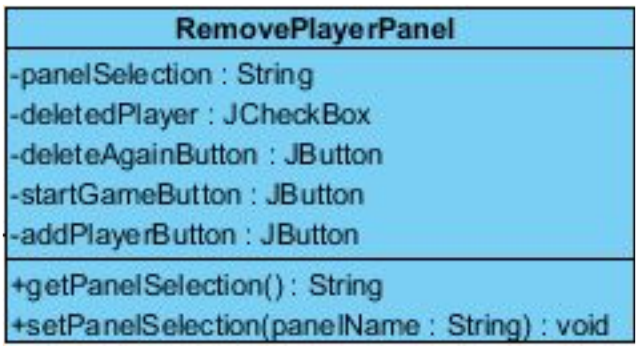
When the player presses deletePlayerButton, String value becomes "deletePlayer" and CanvasView class selects the RemovePlayerPanel object from the cardLayout and showed in the screen. In this panel, there will be JcheckBox object for each player. Selected player will be deleted.

There are also two more Jbutton objects. One of them is addPlayerButton. When the player presses addPlayerButton, String value becomes "addPlayer" and CanvasView class selects the AddPlayerPanel object from the cardLayout and showed in the screen. And all process will be completed as indicated above. The other JButton object is startGameButton and it will be activated after deleting and adding player operations. If player presses startGameButton, String value becomes "startGame" and

CanvasView class gets this information by using getting method. Therefore, CanvasView class selects the GamePanel object from the cardLayout and showed in the screen.

4.4.1.8. EndRoundPanel

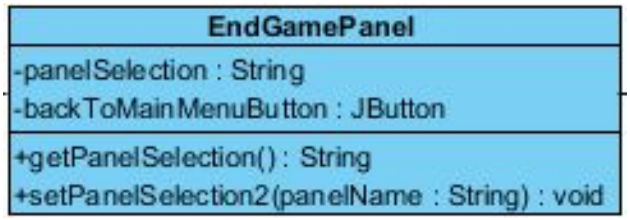
Creates the last discontinued round's panel which shows that each player's score in that round.



When the round is finished, String value becomes "endRound" and CanvasView class get this information by using getting method. Finally, EndRoundPanel object which shows the scores each player gained will be selected from the cardLayout and showed in the screen.

4.4.1.9. EndGamePanel

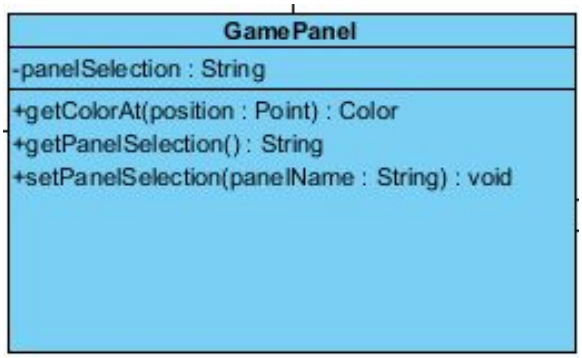
Creates the completed game panel which shows that each player's overall score in the game.



When the game is finished, String value becomes "endGame" and CanvasView class get this information by using getting method. Finally, EndGamePanel object which shows the total scores each player gained will be selected from the cardLayout and showed in the screen.

4.4.1.10. GamePanel

GamePanel is an active playing field.

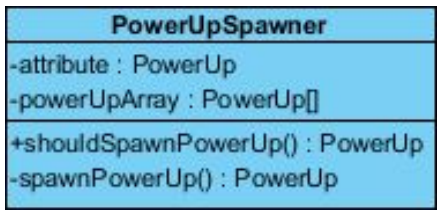


This class works with “GameController” class simultaneously. Therefore, its control will be little bit different from the other panel objects. “GameController” class informs “GamePanel” class for each player’s direction and the game panel will be changed by these inputs. Why do not we use Controller to indicate the direction to “GamePanel”? There are some reasons. First one is about collision detection mechanism. In order to catch collision, “GameController” class checks the coordinate of each player’s head pixel by pixel. That means, “GameController” should take information directly from “GamePanel”. The other reason is about the speed. There are too many instantaneous inputs and outputs between “GameController” and “GamePanel”. If we use “Controller” for each input and output, it makes our game tired.

The beggining view of GamePanel can be created in the control of CanvasView class. When the game starts, GamePanel object is changed by GameControl class, because there will be some instantaneous variables which affect GamePanel object.

4.4.2 Controller Classes

4.4.2.1 PowerUpSpawner Class



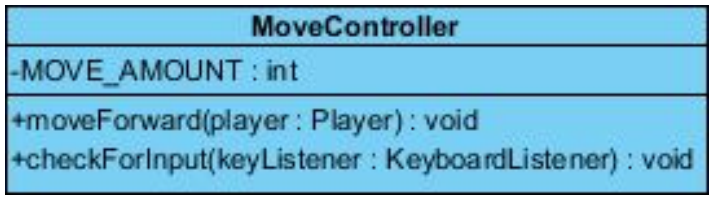
The main purpose of this class is to spawn power-ups at random times and locations in the game. In order to add such a functionality to the system some attributes and methods were added to this class. The "powerUp" attribute should hold the power-up type that is planned to be spawned. It should also change its value when another random power-up type is selected. The "powerUpArray" should hold all of the power-up types. This array will be useful when initializing the powerUp variable when randomly picking a power-up type. Other than the attributes, "shouldSpawnPowerUp" decides whether to spawn a powerUp or not. This method will be public and the GameController class will repetitively call this class to decide spawning a power-up. If this method decides to spawn a power-up, then it will call the "spawnPowerUp" method which is a private method inside this class. This class will randomly select the type and location of the power-up using the powerUpArray and powerUp attributes. Then it will return the power-up it created to the shouldSpawnPowerUp method. Finally, shouldSpawnPowerUp method will return the power-up to the GameController class and the power-up will be created and displayed. On the other hand, if the shouldSpawnPowerUp method decides not to create a power-up then it will return null to the GameController class.

4.4.2.2 CollisionDetector Class

The main functionality of the CollisionDetector class is to decide whether a player has collided with something or not. In order to do so it has some predefined and changeable attributes. For example, the "BACKGROUND_COLOR" attribute is a predefined one which holds the color of the background. However, the other variables can change from time to time in the game. The "playPosition" attribute holds the position of the player as a point and "playerColor" attribute holds the color of the

player to compare it with other colors like the background color. These two variables are initialized from the Player object which is a parameter to the “checkForCollision” method. The last attribute “powerUpColors” is an array which holds the colors of all of the power-ups. “chedckForCollision” method is a public method which decides whether a player has collided with something or not. It gets a Player object as a parameter and returns an integer which resembles different cases of collision. This method, compares the point (playerPosition+1) with the other possible colors in the game. If this point is same with the background color then it decides that the player did not collide with anything. However, if the color of this point is something else, then it checks what color that it matches to and returns the current situation.

4.4.2.3 MoveController Class



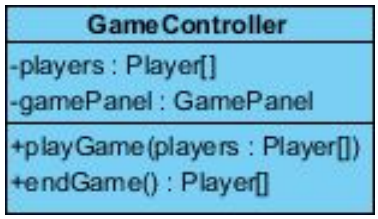
The main functionality of the MoveController class is to move the player forward at each second and to detect whether a player wants to change its direction or not. There is a constant attribute in this class called “MOVE_AMOUNT” which resembles the amount of movement forwardly at each second. Other than that, this class has two public functions that are been called by the GameController class. “moveForward” method gets a Player object as a parameter and does not return anything. The function of this method is to move the player forwardly. The second method is called “checkForInput” and it simply checks the keyboard inputs and behaves according to it. In order to do so, it uses KeyboardListeners. this function also does not return anything.

4.4.2.4 RoundEndController Class

The RoundEndController checks whether the round or the game has ended or not. To add such a functionality to this class some attributes and 2 methods are used. The “numberOfPlayers” variable holds the currently number of players in the game.

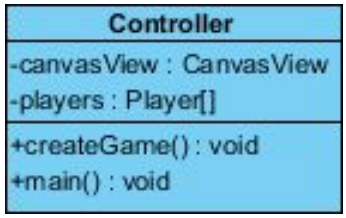
This variable is useful when calculating the winned points of each player after each round. The “scoreNeedToEndGame” is also a variable that is being changed by executing an algorithm using the value of numberOfPlayers. Other than the attributes, the “checkForRoundEnd” method is a public method that gets a player array as a parameter. It checks the current point of all of the players and decides whether the round has ended or not. This method will be called from the GameController class and will return a boolean to indicate the result. Finally, the “checkForGameEnd” method will check whether any player has reached scoreNeedToEndGame variable or not. It also will get a player array as a parameter. This method will be again called from the GameController class and will return a boolean to indicate the result.

4.4.2.5 GameController Class



The GameController class is the main class that is been responsible from the inside game rules, conditions and cases. In order to do so, it works together and keeps contact with the other controller classes explained above. It has a player array called “players” which holds all of the current player objects that are in the game. This player array will be also used in other controllers and will be sent as a parameter. The “gamePanel” attribute will hold the GamePanel instance and will update it according to the gameflow. Other than the attributes, this class has a public method called “playGame” which handles all of the gameflow and is called by the Controller class. Inside this method there will be a loop which iterates every frame and updates the game variables and the GamePanel until the end of the game. The “endGame” method will be called at the end of the game and will update the game score of every player object.

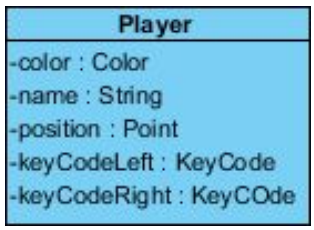
4.4.2.6 Controller Class



The Controller class is the main controller of all of the system. It should have some attributes of instances of other classes in order to keep everything under control and to demand some works from other classes. The variable “canvasView” can be shown as an example to this. Other than that, it should hold all of the player instances in an array called “players” in order to send inputs gotten from the users to the GameController and other controller classes under it. It should also have a public method called “createGame” in order to start a game when a user presses the start game button. This method should call the playGame method inside the GameController class. Finally, this class will hold the main function, thus the program will start executing from this class when the .exe file is oped.

4.4.3 Model Classes

4.4.3.1 Player Class



The Player class will be responsible of creating player objects. It will have a default constructor and a constructor which will be able to initialize a player object with its instances. A player will have a color, a name, a position at a time, and a key configuration thus, the attributes of this class should be according to it. The key configurations, which are the keyboard buttons a player will be able to move to left or right when pressed, will be saved as KeyCode objects and will be used in the MoveController class.

4.4.3.2 PowerUp Class

The class called PowerUp will be responsible of creating power-up objects. It will have a default constructor and a constructor that initializes a power-up object according to its attributes. A power-up will have a color, a position at a time, a type, and a lifetime thus, its attributes should be according to it. The type of a power-up will be stored as a string which can be a number of predefined names. Other than that, the lifeTime of a power-up is the current time left until the power-up demolishes. Initially it will be a constant number which can vary from type to type and will decrement each second. When this variable becomes 0, the power-up should be destroyed.