Bilkent University
CS 319
Object-Oriented Software Engineering

Fall 2014

ELEMENT WARS PROJECT
Final Report

25 December 2014

GROUP #8 Members

Umut Hiçyılmaz

Selçuk Gülcan

Serdar Demirkol

1.Introduction

We are group8 of CS319 course. We name our course project as ‘Element Wars’. It is a turn based collectible card game (CCG). Collectible card games, also known as trading card games, are card games which uses set of cards specifically designed for a game. If they are compared with classical card games like solitaire, poker etc. these classical card games uses the same deck, generally consist of 52 cards. However, cards of a CCG game is unique for that game, another CCG game uses another set of cards. Also, players use the same deck in classical card games whereas every player in CCG games has its own deck. For example, if a CCG game is played by 2 players then there are 2 decks in the play.

Element wars game is played by two player: A human player and an AI player. Player plays against the AI. Every player has a castle and castles have durability point representing players’ health. The goal of the game is to reduce opponent's castle durability to zero.

In this Final report, firstly problem statement will be explained to the reader. Then there is requirement analysis part which includes functional and nonfunctional requirements, constraints, scenarios, use case models and user interface. After that, there is analysis models section which has object model and dynamic model parts. The main parts of the report are system design and object design will be followed in next sections. There are design goals, subsystem decomposition, architectural patterns, hardware/software mapping, persistent data management, access control and security, global software control and boundary conditions sections in the design part and design patterns, class interfaces and specifying contracts using OCl sections in the object design. Report will finish with a conclusion which is about summary of what we explained and lessons that are learnt during the entire project.

2.Problem Statement

As computer engineering students, we are generally interested in pc games when it comes to entertaining. Designing a game for our project has given us a chance to actually build a game such that we could decide how it is going to be and as a result have lot of fun. Designing a game with the result of something that you will like to play.

First of all, as a group we agreed that we can make a collectible card game which is a new game genre becoming popular and easy to customize. Even though playing routine of the game would be similar to the other games in this genre which are already exist, each of them have their unique parts. We thought that we can take the properties that we like from the games and choose what to include or not. This has given us a great chance to actually create something which we love playing and never think what this could be better if it was done another way.

Playing the game is really easy but also challenging. Easy to use user interface is not going to be complex but also satisfying enough to make the game playable. The name of the game is “Element Wars” because of the nature of collectible card games require categorization of the cards. We tought that main elements would do the job perfect.

We tried to make a game which people will like playing it. We used Java to make it available across different platforms. This game can be extended to make it available to play through web browsers.

3.Requirement Analysis

3.1.Functional Requirements

3.1.1. Play Game

This function is required and the most basic feature of the game for the players is starting the game. The user selects the play game button and Element Wars and player tries to beat to computer by playing card in his/her deck. If player didn't create a deck before, s/he starts the game with the default deck which is provided by the game.

3.1.2. Settings

Settings option is the part of the main menu where players can change the external effects of the game manually. For instance, user can change the background color. User can also change the back image of the cards. In addition, if player wants to use the deck s/he created before on other device, s/he can import it by choosing the file as well as export the deck that s/he created on this device. Player can also go back to the default settings.

3.1.3. Settings

Since user needs a deck of cards to play the game, s/he would create a deck to play with his/own choice of cards rather than the default one. In order to create their custom deck, all of the cards on the game will be displayed to the player. Player will choose a certain number of cards to create his/her own deck. Just like creating a new deck, player can delete a deck that s/he created before. If player want to change some cards on her/his deck, s/he can also edit their remaining deck.

3.1.4. Help

Help option provides the necessary information about the Element Wars game. Users can find the rules of the game. Also instructions about how to play the game. This is an important function for new users who doesn't know how to play the game. Also old users can check the rules again to be sure.

3.2.Non Functional Requirements

3.2.1. Performance

Since the game is initially designed as single player game, there is no reason to make the player wait. Game menu transitions will not have long animation delay. Moreover, the game is played against an AI so AI should take its decisions quickly. However, right now it is not possible to predict this decision time. Design process of the project probably helps us at this point later. In short, game should respond fast enough to not make players bored.

3.2.2. Easy to Use

Element Wars is a hard game by its nature. Even without necessity to know cards, players have to know fundamentals mechanics, tradeoffs and making right decisions. If we also consider card effects, it's quietly difficult game to master. Therefore, other parts of the system should be as easy to use as possible in order to lighten this learning phase load. Graphical user interface should be prepared as simple. Players should be able to navigate any page they want by clicking a few buttons. Also, cards itself should be explanatory. Players should know effect of cards even if they don't play those cards before.

3.2.3. Robustness

The game has great replayability. A small changes in decks can create various gameplay. Addition to deck changes, adding new cards into the game collection provides enormous variations. Therefore, the game should be designed carefully so that adding new cards, effects or mechanics should not affect the cards that exist already.

3.4.Constraints

Program will be implemented using Java. Since it is a highly convenient programming language for Object Oriented approach and its performance is satisfactory, we decided to use Java. In addition, Java has many different supporting libraries that will make the coding easier than some other programming languages.

3.5.Scenarios

➔ Play the Game

Alex is already started the game and made his selection of deck. Game has already started and it is Alex's turn. He will chose a card to attack his enemy. He chooses a card from his current hand to play. Then the card that he chose gets checked if there is enough energy to play the card or not. After the control of energy, if the card is playable Alex needs to choose a character to attack with. He clicks on a character to attack with on the screen. Just like choosing a character to attack with, Alex also need to choose a character to target. Of Course both attacker and targets needs to be checked if they are valid or not. Combat is performed if every entry is true. Then combat results are applied on the screen and it is AI's turn.

➔ Create Deck

Alex executes the program, he sees the opening window of the game, then he chooses Deck Manager to create his own custom deck. On the next screen he clicks on “create” button to create his own new deck. All of the cards are displayed to him to pick certain number of them. Game will keep asking him to pick if he hasn't choose enough. Then the number is satisfied, he slicks on save button to keep his deck.

➔ Get Help

Alex executes the program, he sees the opening window of the game, then he chooses the help document to read. There will be explanations of how to play the game and also the rules incase anyone wants to check them or it will be very beneficial for the first time users. He clicks close to go back to the main menu.

➔ Settings

Alex executes the program, he sees the opening window of the game, then he chooses settings to do operations about decks. There are import and export deck options which he can choose. In order to retrieve a saved deck from the disk, he chooses import. After that, using file manager, Alex will choose the deck and it will be imported. To save his custom deck to the disk, Alex needs to choose export option. On the export option, he will choose the place of the deck that he want to save into. He can go back to the main menu after all operations.

3.6.Use Case Models

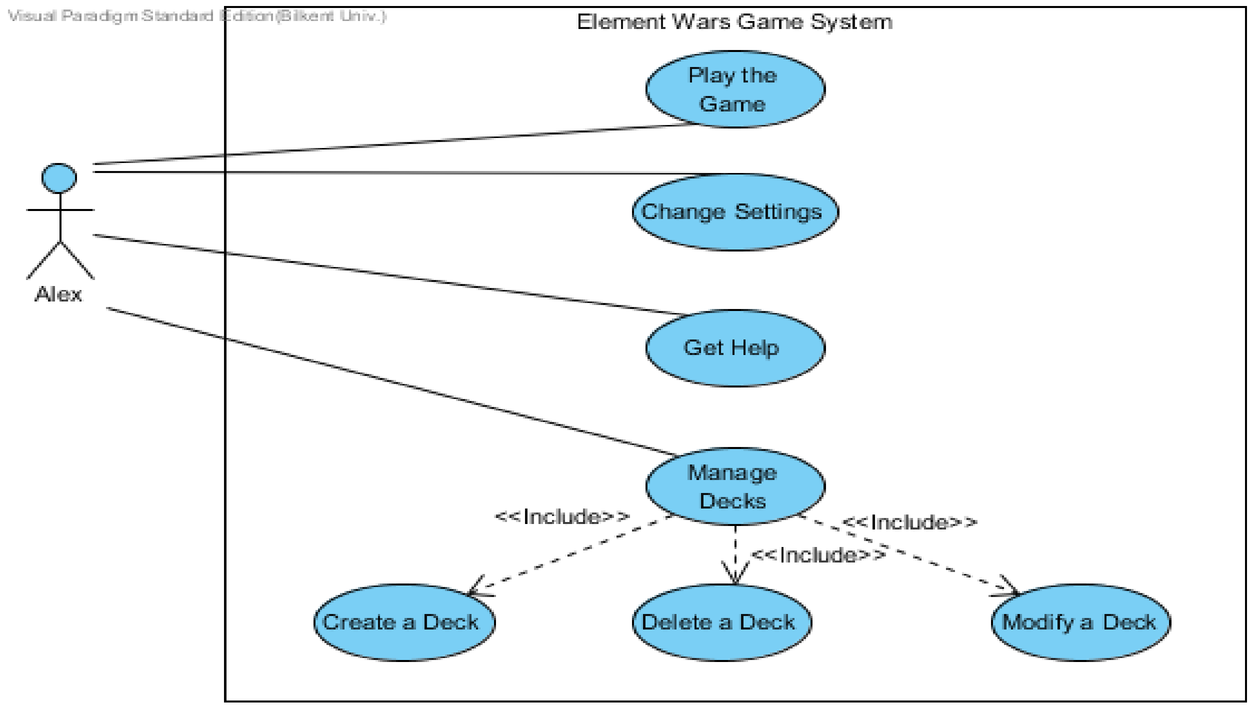
➔ Play the Game

Scope: Element Wars
Level: Subfunction
Primary Actor: Player
Stakeholders and Interests:

- User: Wants to start and play the game..

Precondition: Player has already opened the game and is on the main menu.

Success Guarantee: Player plays the game and the game is over, result screen appears.



Main Success Scenario:

1. Player selects "Play Game" option in the main menu, deck selection screen appears.
2. The player selects the deck he/she wants to play with and then press "Start the Game" button and the game starts.

Extension:

A. Player changes his decision and wants to return the main menu. Clicks "Back" button on the screen and then returns the main menu.

3. The game gives each player a specific number of cards from their decks.
4. Coin flips and starting player is chosen randomly.
5. Turn starts and the owner of the turn draws a card from his/her deck.
6. Beginning phase of the turn starts and any related effect resolves.
7. Play phase starts. If it is AI player's turn, it plays its turn by looking its predefined rules. If it is player's turn, he/she also play his/her turn.

A. Player chooses to not play anything and clicks "End Turn" button.

B. Player chooses to play cards and clicks the card he/she wants to play and then clicks "End Turn" button.

1. Player has enough energy crystals to play the card and card is played on the battlefield. Any effects related to this action resolve.
2. Player does not have enough energy crystals to play the card and a warning message appears.

8. Player also may use his characters on the battlefield to attack.

A. Player should first choose which character he wants to attack with.

B. Then choose an enemy character he wants to attack.

Extension:

    1. If the player chooses a wrong target such as a friendly character then a proper warning message appear and game expects the player to choose a valid target.

8. After all actions play phase ends, end phase starts. Any related effect resolves.
9. Turn ends, it passes to the other player.
10. Turns are passed until one of the players' health reduces to zero or below.
Extension:

A. Before players' health reduces to zero, all of the cards in one of the deck are depleted. If so, the character whose deck is empty loses the game and game is over.

11. The game is over. A victory or lose result screen appear.
12. The player clicks "Return to Main Menu" button and main menu appears.

➔ Change Settings

Scope: Element Wars
Level: Subfunction
Primary Actor: Player
Stakeholders and Interests:

- User: Wants to change settings of the game.

Precondition: Player has already opened the game and is on the main menu.
Success Guarantee: Options set the game according to requests of the player. After that, player returned to main menu of the game.
Main Success Scenario:
1. Player selects "Settings" menu from the main menu.
2. A menu containing all options and settings for the game is showed up.
3. Player arranges the settings and options according to his/her choice.
4. User clicks to "Save" button.
5. New setting are applied.
6. Player is returned back to main menu.

Extensions:
1. Player clicks "Cancel" button.
2. Changes are not applied and player is returned back to main menu.

➔ Get Help

Scope: Element Wars
Level: Subfunction
Primary Actor: Player
Stakeholders and Interests:

- User: Wants to learn how to play the game or get information about the game.

Precondition: Player has already opened the game and is on the main menu.
Success Guarantee: Player saw the instructions to play "Element Wars" . Next, player returned to main menu of the game.
Main Success Scenario:

1. Player selects “Help” menu from the main menu.
2. In “Help” menu, controls and instructions of the game is introduced.
3.Player reads and comprehends the instructions and rules, s/he clicks the “Back to Main Menu” in order to return to main menu.

➔ Manage Decks

Scope: Element Wars
Level: Subfunction
Primary Actor: Player
Stakeholders and Interests:

- User: Wants to manage his/her decks.

Precondition: Player has already opened the game and is on the main menu.
Success Guarantee: Player did changes his/her wanted on decks and returned back to the main menu.
Main Success Scenario:
1. Player selects “Manage Decks” menu from the main menu.
2. In “Manage Decks” menu, there are three options listed which are “Create Deck”, “Delete Deck” and “Modify Deck”
3.Player clicks one of the options and selected menu appears.

A. Create Deck
   1) All of the card in the games are represented to the player. Player selects a certain number of cards that s/he wants use add to his/her deck.
   2) Player clicks the “Save” button. A deck created from his/her selection and returned back to then “Manage Decks” menu.

   Extension:
   1) User clicks “Cancel” button either before, after or during the selection and returned back to the “Manage Decks” menu.

B. Delete Deck
   1) Player is asked if s/he wants to delete the deck.
   2) Player clicks the “Yes” button. The saved deck is deleted and returned back to the “Manage Decks” menu.

   Extension:
   1) Player clicks “No” button. No changes are made and returned back to the “Manage Decks” menu.

C.Modify Deck
   1) All the card in player’s deck are showed to the player on the bottom of the screen and the rest of the all card are showed on the top.
   2) Player clicks on a card which s/he want to delete for his/her deck and selected card is removed.
   3) Player clicks on a card from the list on the top which s/he wants to add to his/her deck.

4) Selected card is added to the player's deck. Player clicks "Yes" button. Changes made in his/her are saved and returned back to the "Manage Decks" menu.

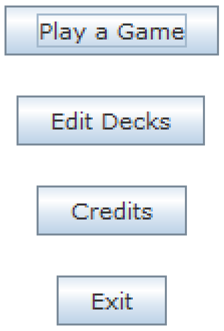
Extension:

1) User clicks "Cancel" button either before, after or during the selection and returned back to the "Manage Decks" menu.

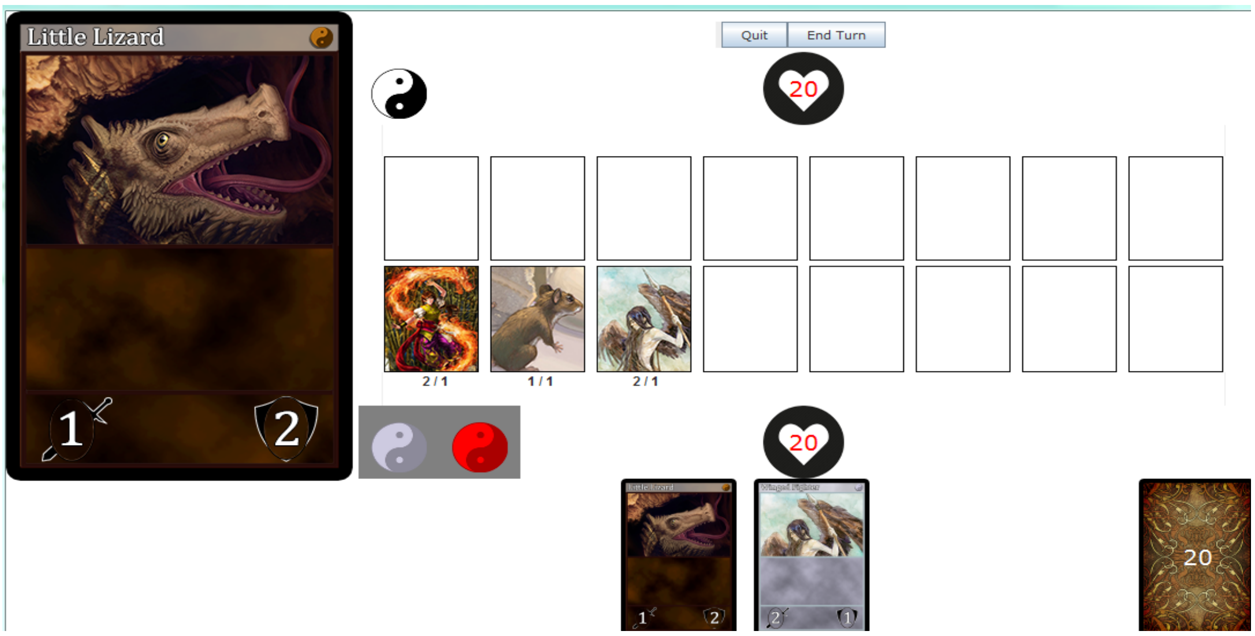
3.4.User Interface

➔ Main Menu

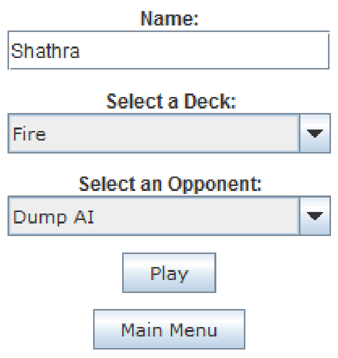
The usage of the main menu of the game is very straightforward. In the main menu, player is expected to choose one of four options: Play game, help, settings and manage decks. Play game button starts a new game and player stays in the game until it finishes. Help button shows some useful information about how to play tips and rules of the game. Settings is the screen that player may export and import deck information so that he/she can share it with someone else or can use deck of other people. Manage deck button opens another menu consists of three options: New deck, delete deck, modify deck.



➔ Gameplay



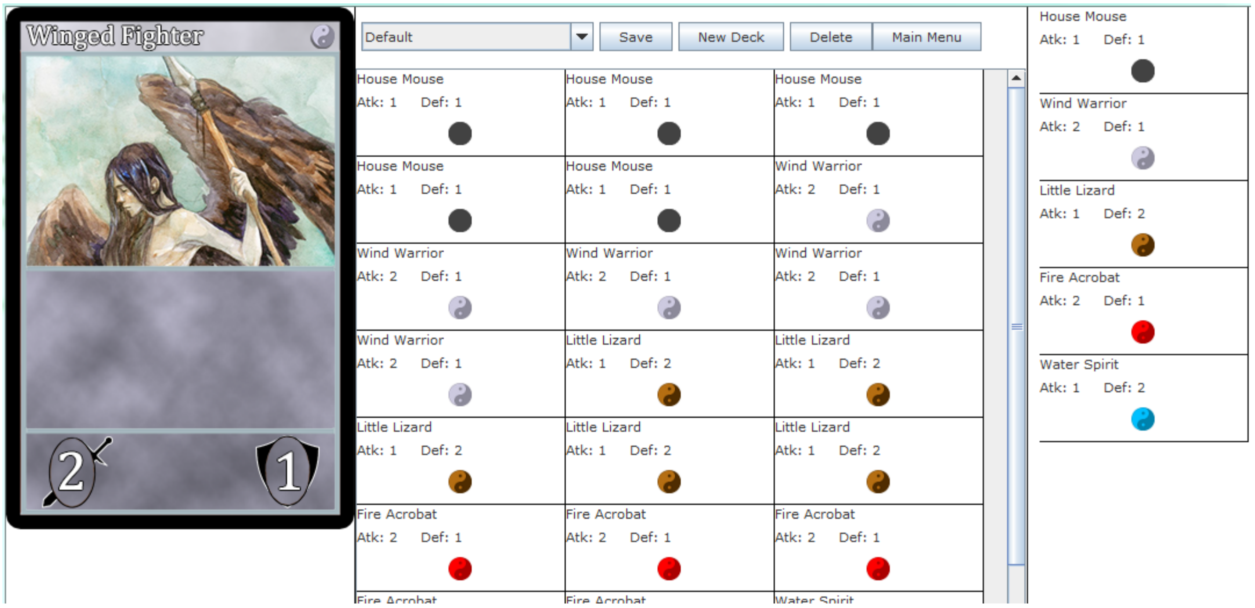
➔ Settings


Name: Shathra

Select a Deck: Fire

Select an Opponent: Dump AI

➔ Manage Decks



House Mouse Atk: 1 Def: 1	House Mouse Atk: 1 Def: 1	House Mouse Atk: 1 Def: 1
House Mouse Atk: 1 Def: 1	House Mouse Atk: 1 Def: 1	Wind Warrior Atk: 2 Def: 1
Wind Warrior Atk: 2 Def: 1	Wind Warrior Atk: 2 Def: 1	Wind Warrior Atk: 2 Def: 1
Wind Warrior Atk: 2 Def: 1	Little Lizard Atk: 1 Def: 2	Little Lizard Atk: 1 Def: 2
Little Lizard Atk: 1 Def: 2	Little Lizard Atk: 1 Def: 2	Little Lizard Atk: 1 Def: 2
Fire Acrobat Atk: 2 Def: 1	Fire Acrobat Atk: 2 Def: 1	Fire Acrobat Atk: 2 Def: 1
Fire Acrobat	Fire Acrobat	Water Spirit

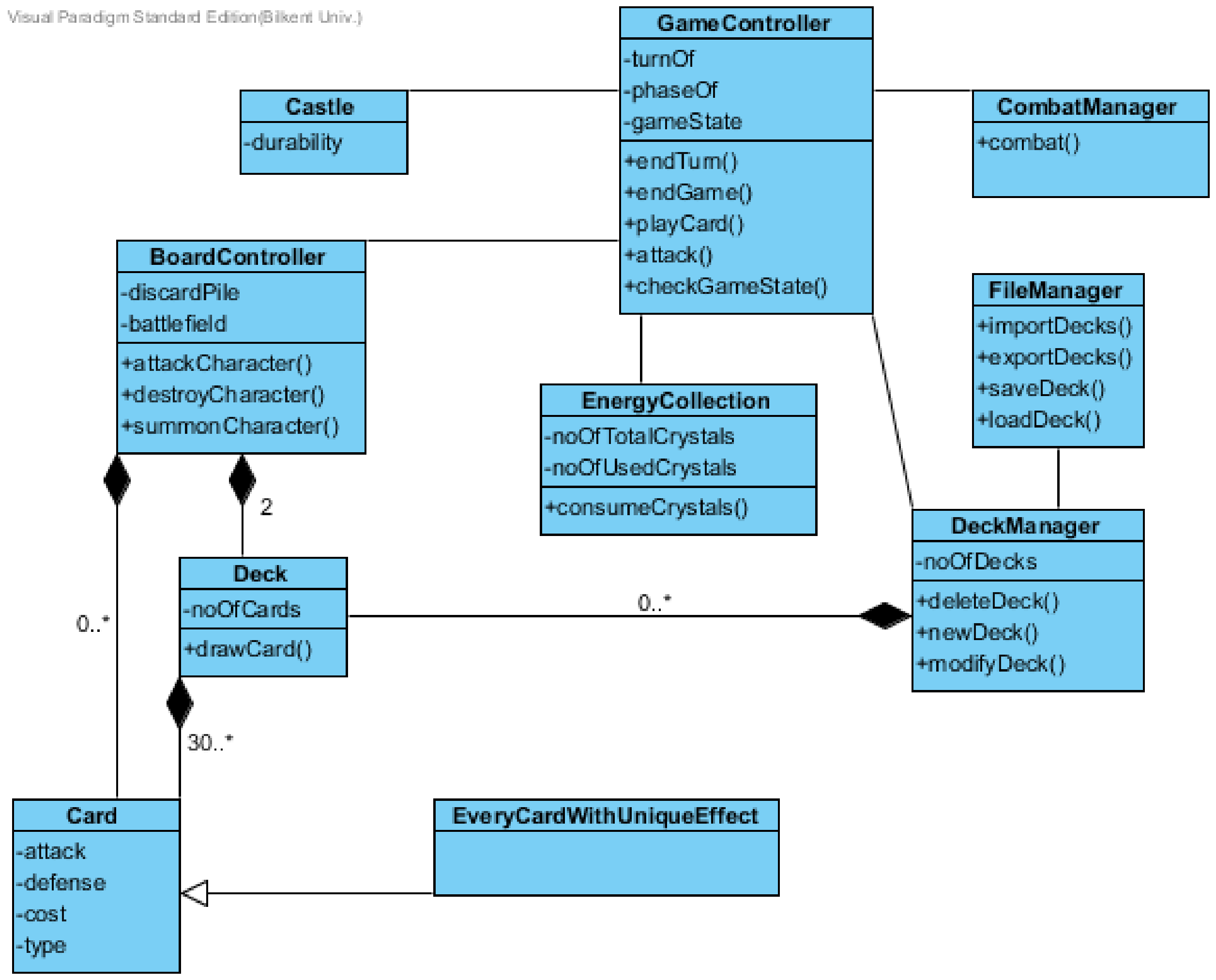
House Mouse Atk: 1 Def: 1
Wind Warrior Atk: 2 Def: 1
Little Lizard Atk: 1 Def: 2
Fire Acrobat Atk: 2 Def: 1
Water Spirit Atk: 1 Def: 2

4.Analysis Models

4.1.Object Model

4.1.1.Domain Lexicon

4.1.2. Class Diagram



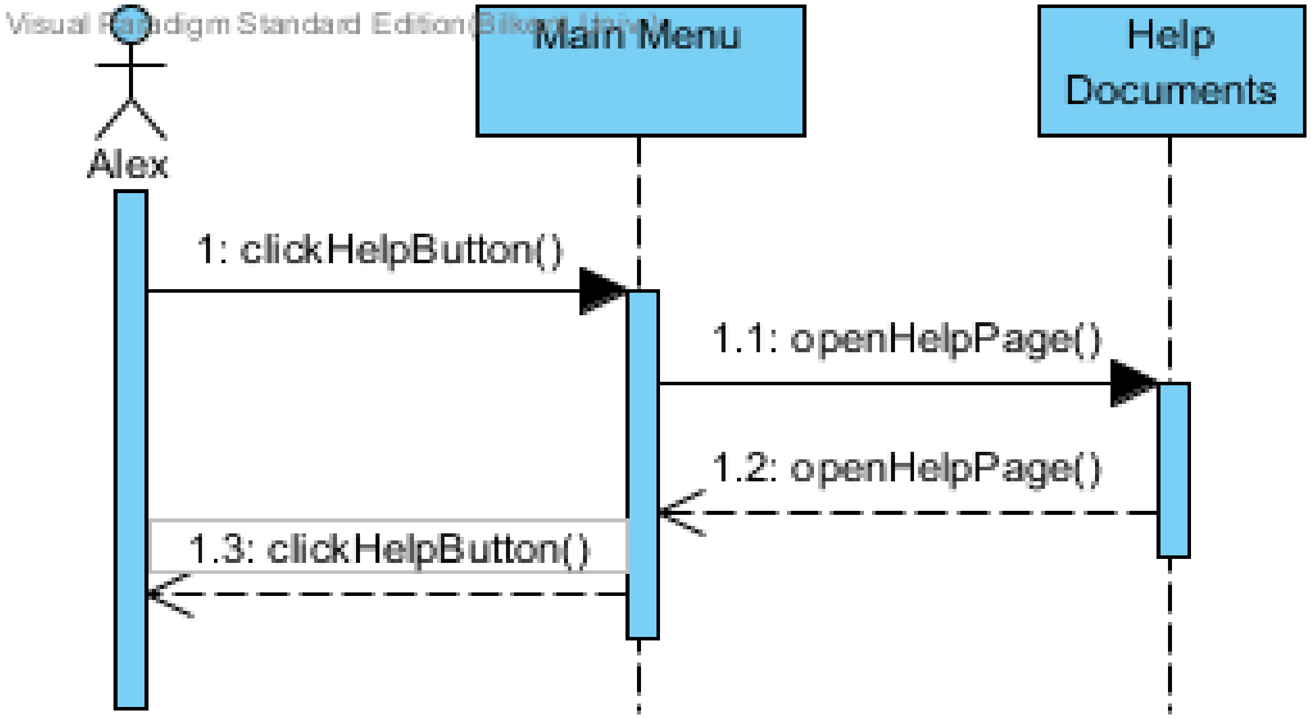
This is the very basic class diagram of the game. Although fundamental classes are represented, many details are omitted in order to keep diagram simple. There is GameController class at the heart of the system. This class is responsible for almost everything about gameplay. Most simplest element of the system is Card class. It presents a single card object, it has basic attributes like attack, defense and cost points. Although some of the cards in the game can be represented by this class, it might be insufficient to express some unique effects cards may have. Therefore, any card that has an unique effect should be written as new classes and these new classes have all attributes of Card class.

4.2.Dynamic Model

4.2.1.Sequence Diagrams

➔ Help

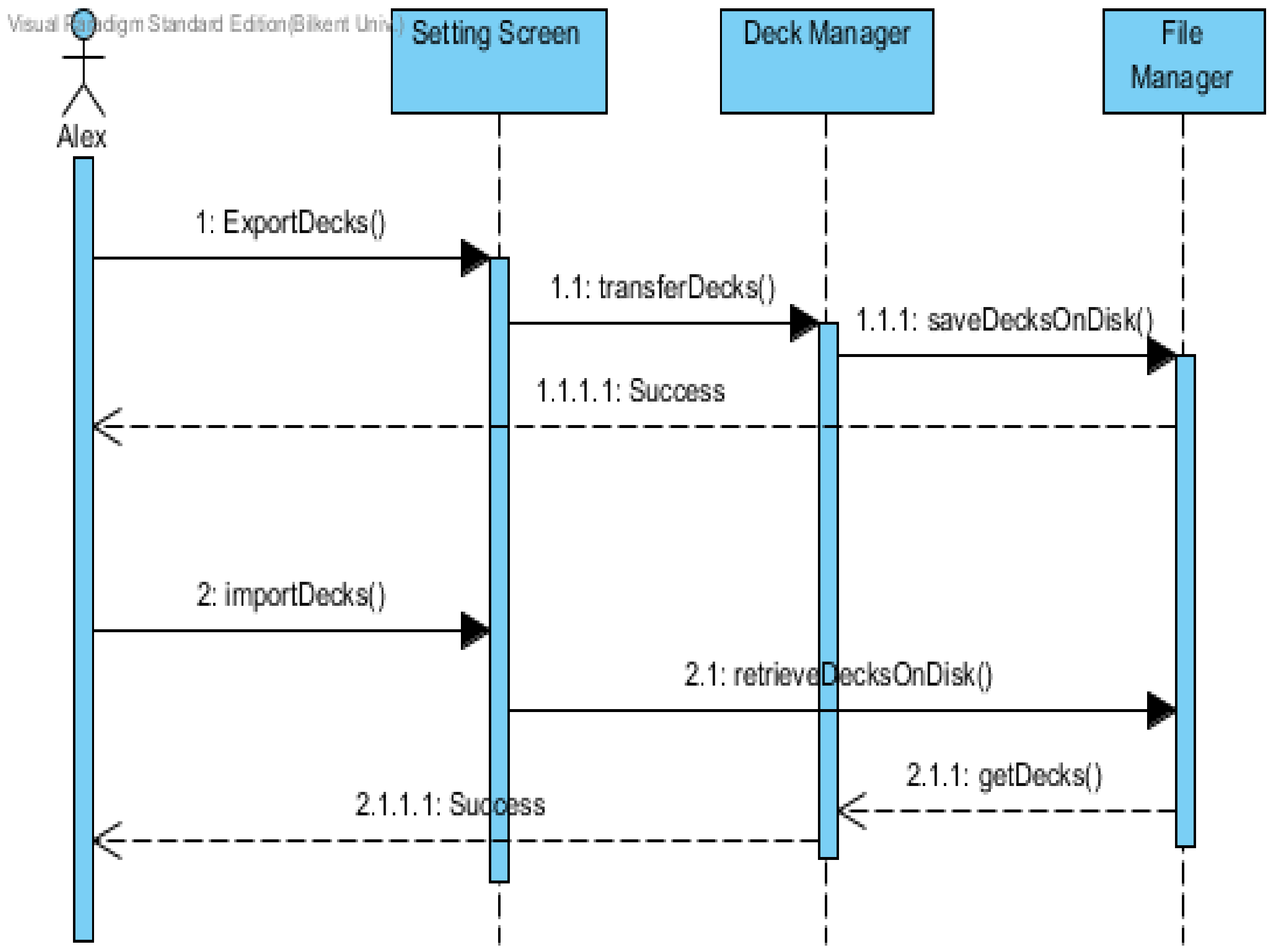
Alex executes the program, he sees the opening window of the game, then he chooses the help document to read. He gets information about how to play the game (i.e. the instructions) and game rules.



This diagram refers to get help scenario. Alex executes the game. Then he requests to see the help document related to the instructions of the game. Main Menu contains the main window of the game.Main Menu is in interaction with the Help Window and its contents. The player sends a request for opening the help window. Main Menu responds to the player and after opening the help window, player can see the help page which contains the related information about the playing instructions and rules of the game

➔ Settings

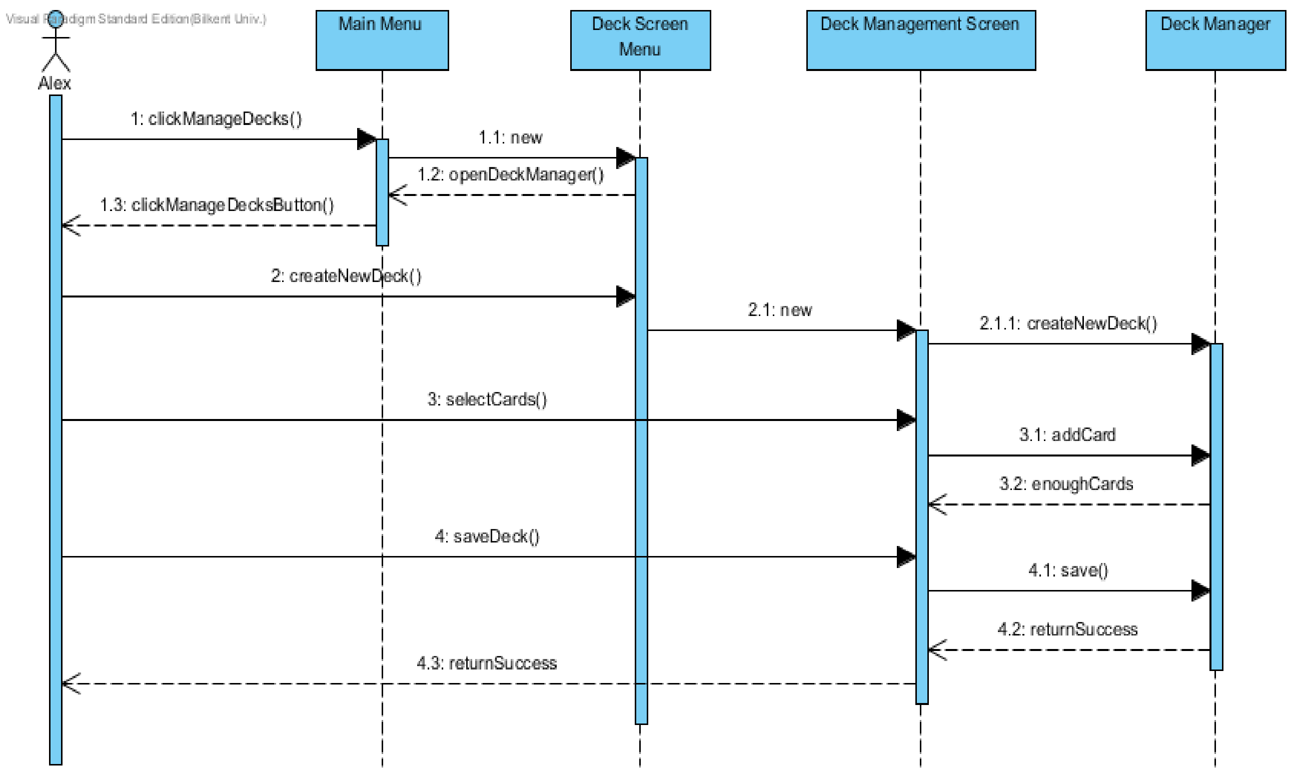
Alex is already in the Settings Menu. First he chooses to export his current custom deck to the computer and the file is saved. Then, he exports a previously saved deck which is already in the computer.



This diagram refers to settings scenario. Alex is already in the Settings Menu. He requests to export his custom deck to the computer. He clicks on “Export Deck” button on Setting Screen and then his request processed by the Deck Manager. Deck Manager gets the request and send the deck to the File Manager which will save the deck on the computer. After the export operation, Alex want to import a saved deck from the computer. He clicks “Import Deck” button on Setting Screen and he is asked to select the file from the computer. File Manager gets the file from its place and sends it to Deck Manager where it will be implemented to the

➔ <u>Deck Manager</u>

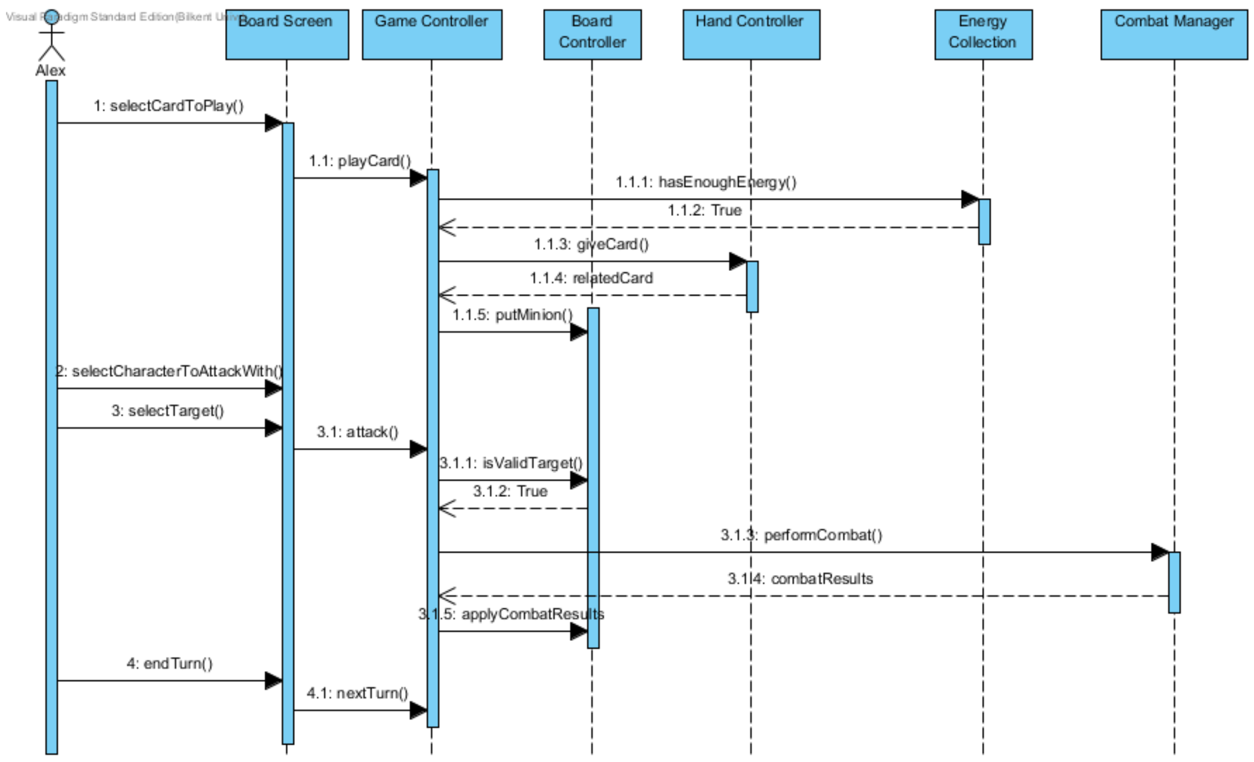
Alex executes the program, he sees the opening window of the game, then he chooses Deck Manager to create his own custom deck.



This diagram refers to create deck scenario. Alex clicks Manage Decks on the main menu then "Deck Screen Menu" is displayed. On this menu, Alex clicks on "Create" button on Deck Screen Menu to create a new deck. Deck Manager creates a new deck to be filled. "Dect Management Screen" is displayed to Alex to let him make his card choices. Alex selects card until a certain card number to complete the deck. Then again on the Deck Management Screen Alex clicks on the save button which leads deck manager to save the deck.

➔ Play the Game

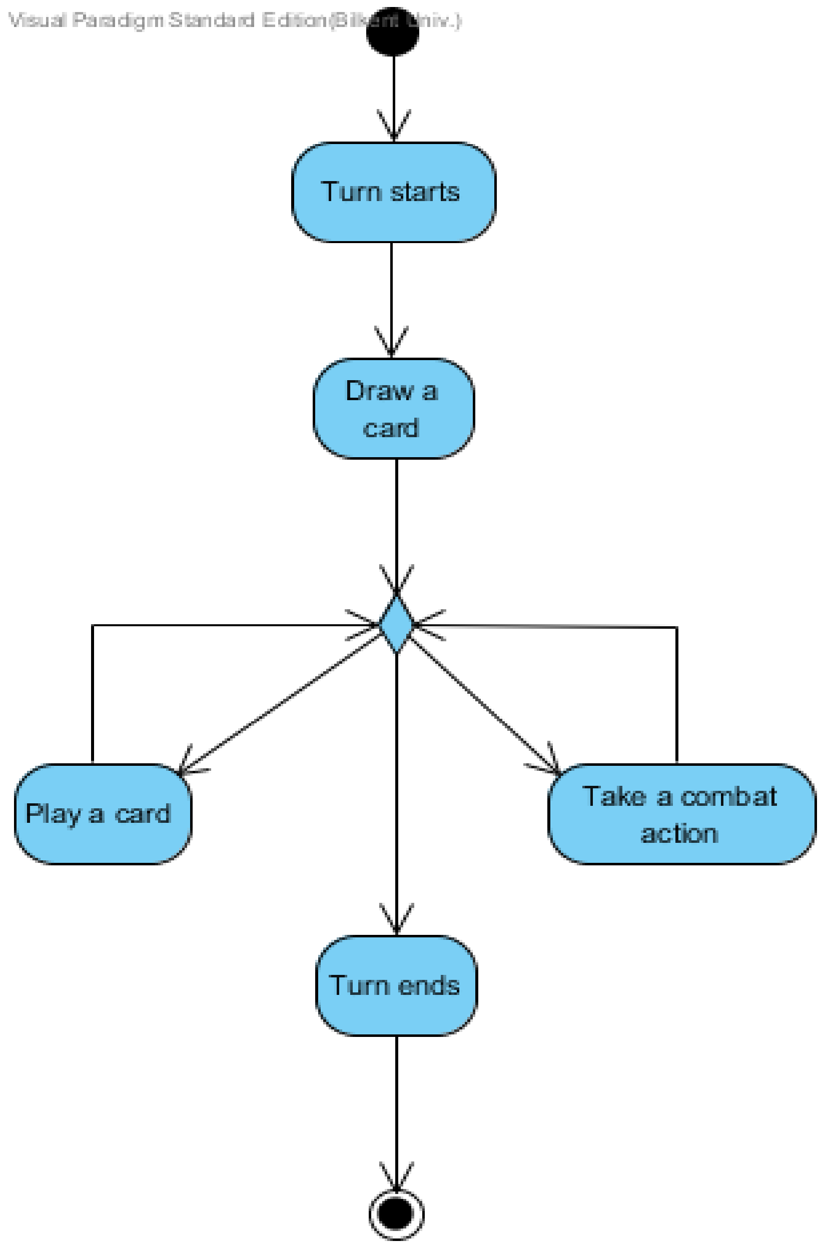
Alex is already started the game and made his selection of deck. Game has already started and it is Alex's turn. He will chose a card to attack his enemy.



This diagram refers to play the game scenario. Alex chooses a card to play at the beginning on the "Board Screen". Then "Game Controller" gets the card and checks if there is enough energy to play the card or. After the control of energy, "Hand Controller" determines card to play and Game Controller puts a minion . Next, Alex chooses a character to attack with on the Board Screen and selects target to attack. Game Controller sends the selected target to the Board Controller to see if it is a valid one or not. Since it is a valid one, combat is performed by the "Combat Manager". Then combat results are applied on the Board center and user clicks to end turn.

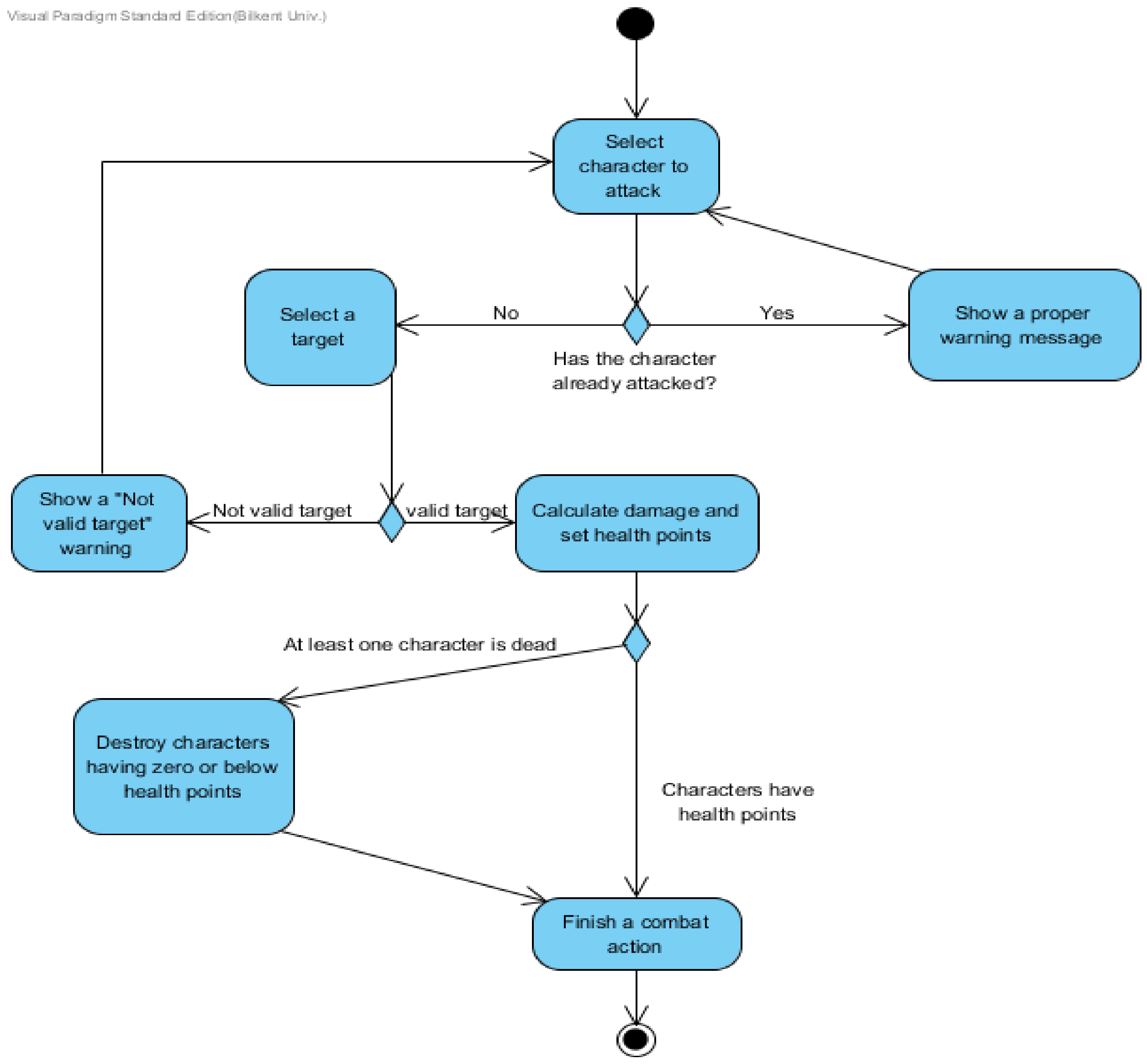
4.2.1.Activity Diagrams

Our game is a turn based game so the main activity is a start of a turn state. After the turn state starts user draws a card from his/her deck. After the draw, player may play card(s), take combat actions or just ends the turn. After the player decides to end his turn, turn ends after the player press "End Turn" button. Then turn ends.



Turn Activity Diagram

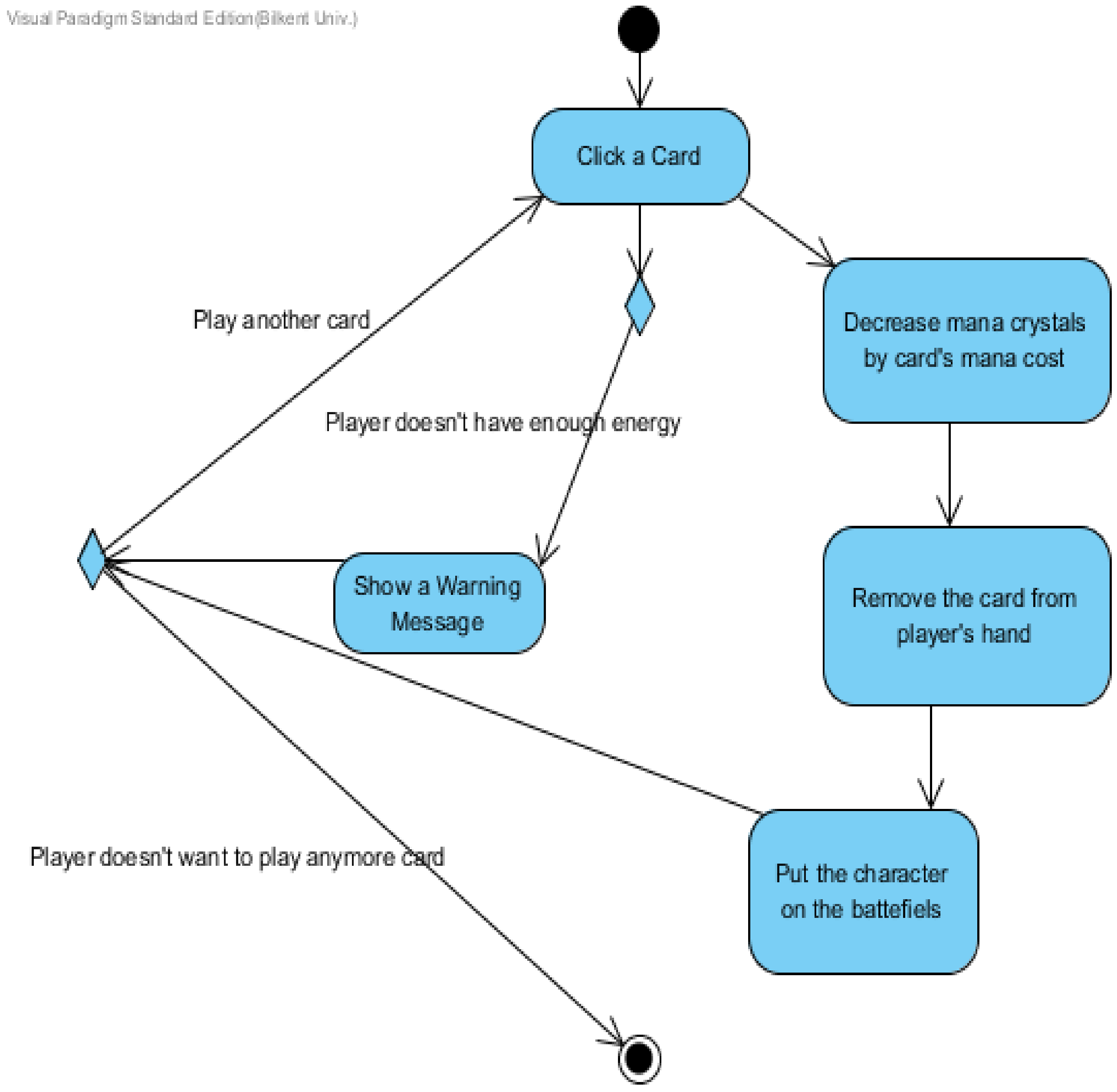
One of the actions the player may perform on his/her turn is to attack. First, player select the character he/she want to attack with. However, each character can attack only once in a turn. So, the character the player chooses has already attacked then the player have to choose another character instead. If he/she manages to choose a valid character then it is expected for player to choose an enemy target. If the player choose an invalid target, a warning message appears and the player should choose another target. If the target is valid, damage calculations are done and by looking at characters' health points, dead and live characters are determined and lastly dead characters are removed from the battlefield and so a successful combat action is performed.



Visual Paradigm Standard Edition(Bilkent Univ.)
Select character to attack
Select a target
No
Yes
Has the character already attacked?
Show a proper warning message
Show a "Not valid target" warning
Not valid target
valid target
Calculate damage and set health points
At least one character is dead
Destroy characters having zero or below health points
Characters have health points
Finish a combat action

A Successful Combat Action Diagram

The other action the player may take is to play a card from his/her hand. Condition of the this action is to have enough energy crystal to meet the card's cost. If this condition is not satisfied, a proper warning message appears and wants the player to choose another card. If the player has enough energy then he can play the card. To do so, the game firstly decreases player's energy, then removes the card from player’s hand and put it into the battlefield. By doing so card play action is completed successfully.


Visual Paradigm Standard Edition(Bilkent Univ.)
Click a Card
Decrease mana crystals by card's mana cost
Remove the card from player's hand
Put the character on the battefiels
Play another card
Player doesn't have enough energy
Show a Warning Message
Player doesn't want to play anymore card

A Successful Card Play From Hand Diagram

5.System Design

5.1.Design Goals

Our main design goal is to reduce complexity. We believe that choosing right architecture style is very important to achieve this goal. For this reason, high cohesion - low coupling will be our design rule. In addition to reducing complexity; clear and satisfying documentation is a goal we would like to achieve. For future changes in the system or other developers, project documentation can be a trustful source.

5.1.1.Ease of Use

Element Wars is a hard game by its nature. Even without necessity to know cards, players have to know fundamentals mechanics, tradeoffs and making right decisions. If we also consider card effects, it's quietly difficult game to master. Therefore, other parts of the system should be as easy to use as possible in order to lighten this learning phase load. Graphical user interface should be prepared as simple as possible. Players should be able to navigate any page they want by clicking buttons. Also, cards itself should be explanatory. Players should know effect of cards even if they didn't play those cards before.

5.1.2.Performance

Since the game is initially designed as single player game, there is no reason to make the player wait. Game menu transitions will not have long animation delay. Moreover, the game is played against an AI so the AI should take its decisions quickly. However, right now it is not possible to predict this decision time. Design process of the project probably helps us at this point later. In short, game should respond fast enough to not make players bored.

5.1.3.Robustness

The game has great replayability. A small changes in decks can create various gameplay. Addition to deck changes, adding new cards into the game collection provides enormous variations. Therefore, the game should be designed carefully so that adding new cards, effects or mechanics should not affect the cards that exist already.

5.1.4.Reliability

The game should be reliable. Since it is one of the fundamental design goals, our system should also reach it too. Game should run without any error and no data loss should occur. Even though it is just a basic game, it should give different result when the same input was given. Otherwise user may not want to play the game again. Reliability requires a strong implementation and testing process but it is aimed to allocate enough time and resources to implementation and testing to reach reliability.

5.1.5.Portability

The program should run on both Linux and Windows machines. This suggests that the program should ensure ease of portability between different environments. Moreover, the decks that are created by the user should be saved into external file. Then it should be possible to use this file in some other copy of the program. User should be allowed to view, change, or edit the decks independent of the program copy used on some other platforms.

5.1.6.Maintainability

The source code should be written in a generic way that will make it easier to understand and contribute the coding process while implementing the project. It is expected to generate a common way of implementation and using this throughout. The documentation of the program should be well-organized to make it easier for new developers to understand the system. The source code documentation, analysis and design reports should be readable and understandable. By that way, developers could contribute new features to the system easily.

5.2.Subsystem Decomposition

In this section, details of system architecture we would like to offer is given. Our final goal in object oriented approach is to reduce complexity. One of the way to reduce complexity is decomposition so we decompose our system into subsystems. While doing this we would like to keep relationship between subsystems minimum and connections between classes in the same subsystem maximum. In order to achieve this goal, we choose an architectural style. This section describes the details about our subsystem decomposition.

As shown in the figure below, our system has three main subsystems which are User Interface, Controller and Game Entities. They are working on completely different cases and they are connected each other in a way considering any change in future. Our main purpose while decomposing our system is to reduce complexity. We try to reduce complexity by high cohesion - low coupling principle.

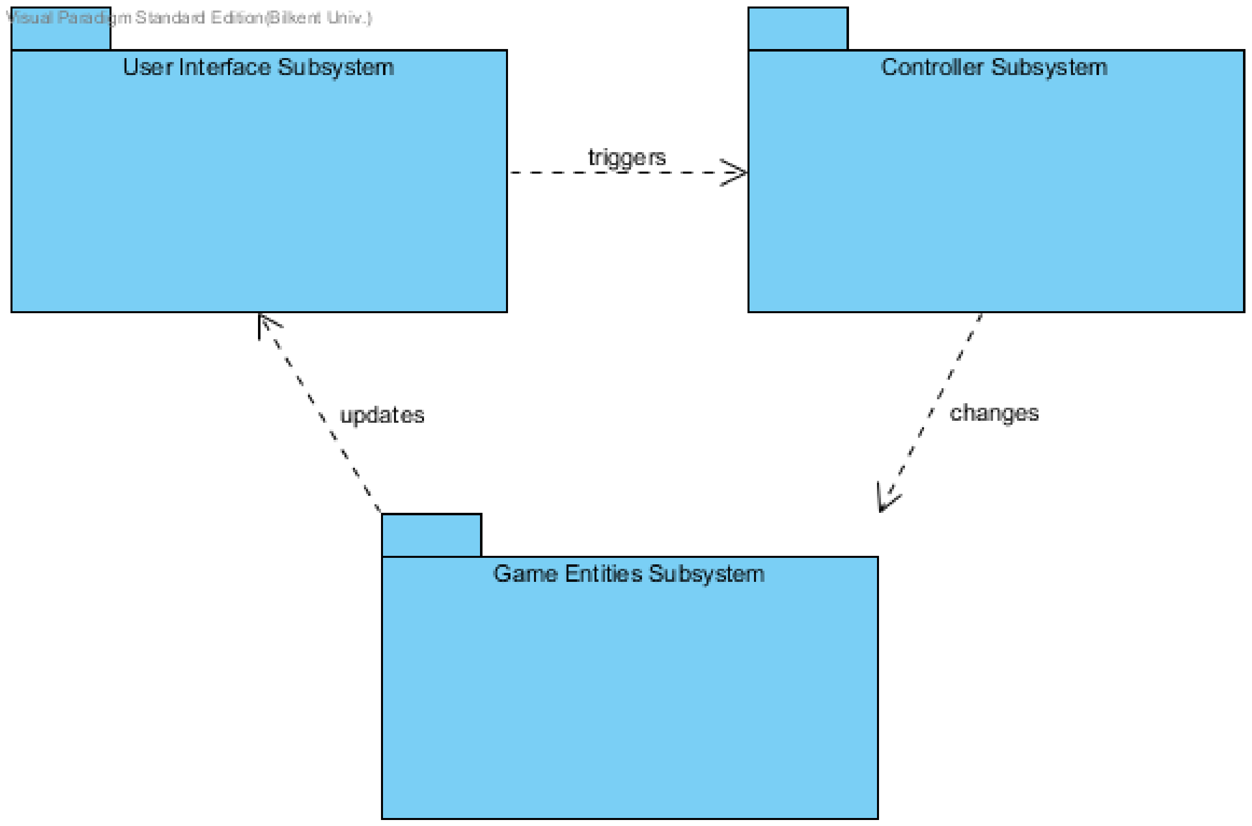


Figure 1 - Subsystems

User interacts with User Interface subsystem. Choices made by user is recognized by the user interface. After User Interface takes the actions, it sends these actions to the Controller subsystem. Controller subsystem checks whether they are valid or not. If they are valid, Controller subsystem calculate the outcome of the actions by communicating with game entities subsystem since game entities hold all data about game entities. Then, according to the outcome, state of the game entities are changed. Lastly, game entities updates corresponding view elements.

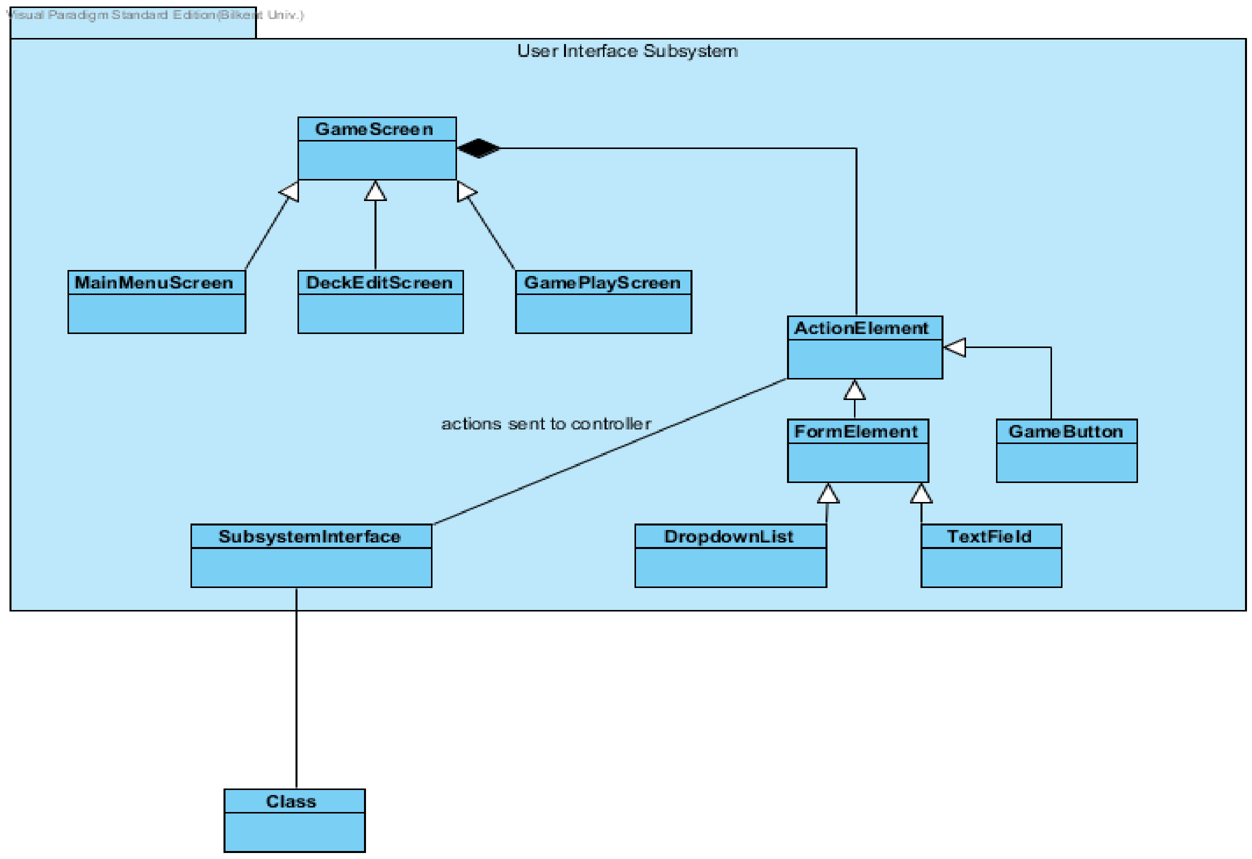


Figure 2 - User Interface Subsystem

User interface subsystem includes all game screens and action elements in game. All these game screens are kind of GameScreen. GameScreen may contain many ActionElement objects. These objects control user interaction. For example, users should click the “End Turn” button in order to finish their turn. Text fields, drop down lists and other form elements are other form of ActionElement object.

Any action captured by ActionElement objects (e.g. clicking a button, filling a text field) is conveyed to SubsystemInterface. This class is responsible the communication between user interface subsystem and controller subsystem. We create this class in order to reduce coupling.

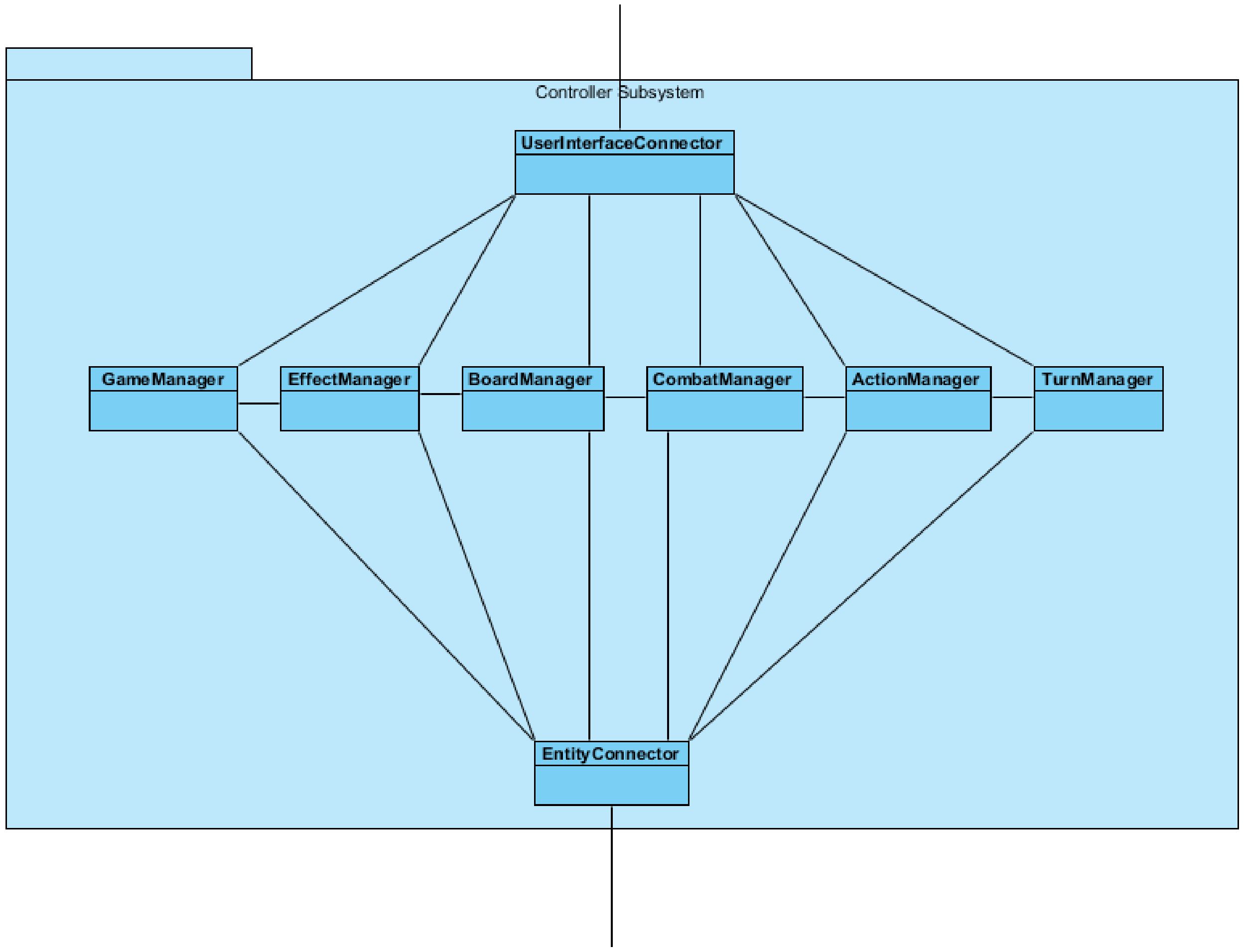


Figure 3 - Controller Subsystem

Controller Unit receives actions from the User Interface, identifies them and sent to proper manager object shown on above diagram. There are many manager classes in this subsystem because controller subsystem is the logic unit of the system. It manipulates the game data stored in game entities subsystem. Therefore, It has to get information entities and call change methods of the entities so it has to connect with entities. EntityConnector takes role in this step, it controls the data flow between these subsystems. Otherwise, there will be many connection out from subsystem, which may lead high coupling and complexity.

These manager objects are also connected to each other because every object's control range is different and action evaluated in a manager object may result in manipulation which is controlled by other manager object. To give an example, A character on the board attacks an enemy character. This may seem an issue of CombatManager class but if combat between these two characters result in death, dying character should be removed from the board. However CombatManager cannot control the board so it should report this situation to BoardController so that it can adjust the board according to combat results. This kind of relationships cause many connections between manager classes and little connections between controller subsystem and entity game system. This is what we want: low coupling, high cohesion.

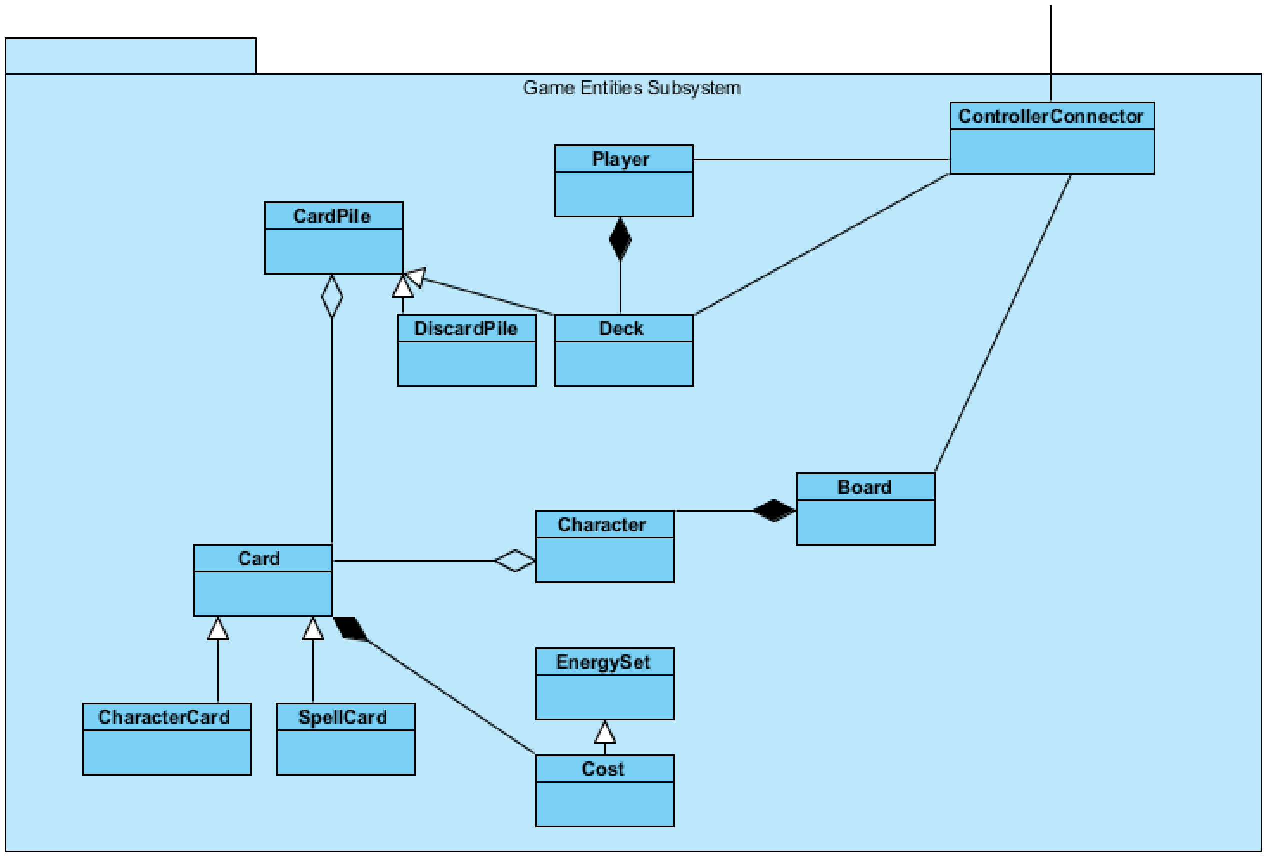


Figure 4 - Game Entities Subsystem

Game entities subsystem contains all game model classes such as card, player, deck, board, character, spell and so on. These all are described in detail later on this report. What we would like to emphasize is connector class and views. As in other subsystems a connector class is used to communicate with other subsystems. Also, entity subsystem is the only subsystem which changes user interface. User interface graphics is a representation of entities. Whenever a state of an entity changes, corresponding view is updated.

However, after that we realized that we can divide Game Entities Subsystem into other subsystems to reduce complexity even more. ControllerConnector should depends on Player and Board but Player and Board do not have to know anything about ControllerConnector. The same relationship exists between Player-Board and Card-Effect-Character. Board depends on Card but Card does not. These hierarchical relationships creates such a subsystem decomposition figured below.

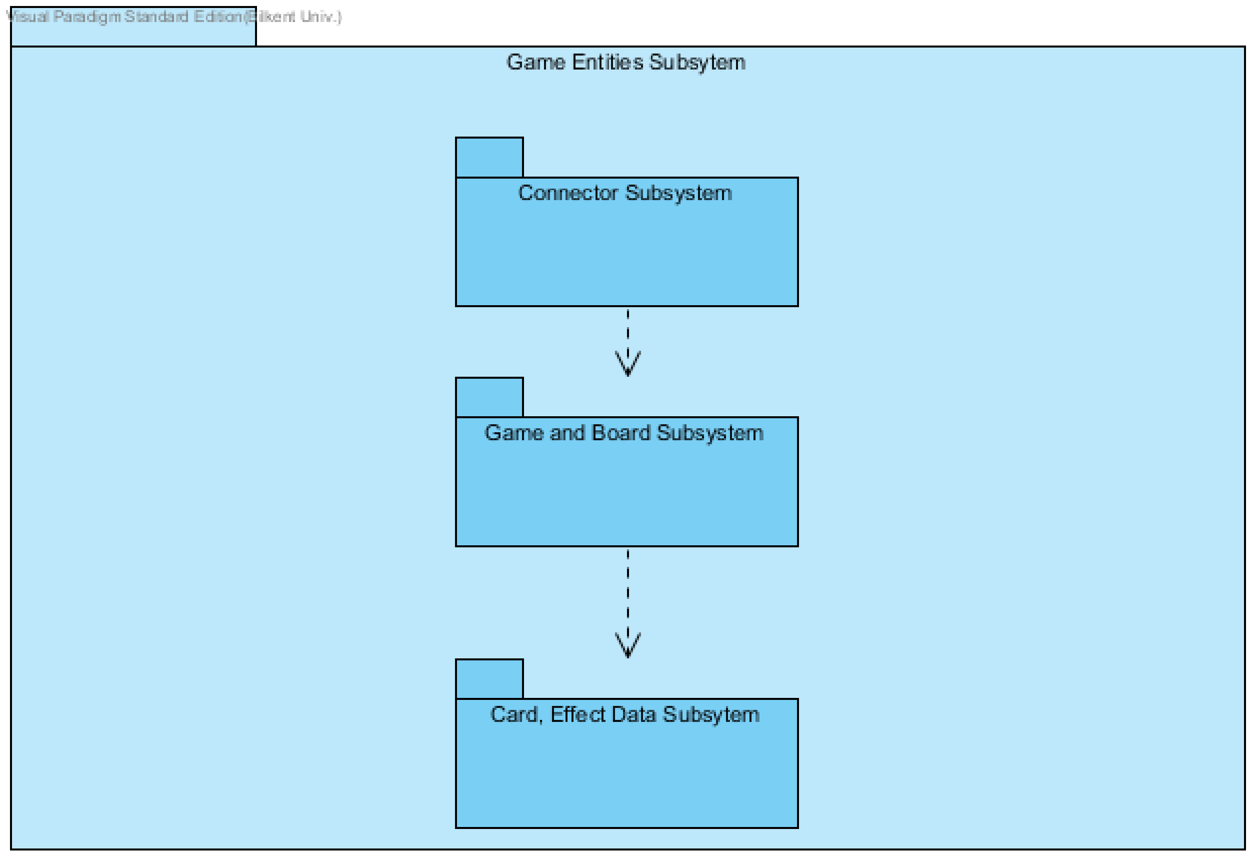

Visual Paradigm Standard Edition(Bilkent Univ.)
Game Entities Subsytem
Connector Subsystem
Game and Board Subsystem
Card, Effect Data Subsytem

Figure 5 - Game Entities Subsystem Decomposition

There is a direct relationship between our subsystem decomposition and architectural style choices. Next section describes these architectural styles we use.

5.3.Architectural Patterns

5.3.1.MVC

In figure 1, the whole system is divided into 3 subsystem. Game entities Subsystem is responsible for holding information of game objects whereas controller subsystem is the logic unit and user interface subsystem is the view of the program. There is a triangular relationship between subsystems. These situations support that this is MVC architecture. Game entities subsystem is model, controller subsystem is controller and user interface subsystem is view. Why we use this architectural style is that MVC architecture provides better encapsulation. Since game data, game logic and user interface are different cases, dividing them can be good idea. Also, the relationship between model-view-controller perfectly suits our subsystem decomposition. User interface (view) triggers controller to take action, controller manipulates game entities (model) and game entities updates user interface.

We believe that MVC architectural style will help us during implementation stage of the project. The separation of user interface and game logic allows us to change interface or game logic without affecting the other. Thanks to this advantage, two different developers can work on different parts of the project without having any difficulty. This issue is also addressed in section 6.5 Management Intractability.

5.3.2.Layered Architecture

The other style we will use is layered architecture. We don't use it as main architectural style but we use it as architectural style of game entities subsystem. Consider figure 5, connector subsystem, game-board subsystem and card-effect data subsystem form a layered architecture. In layered architectures, data flow is from bottom to up and top layers depend on bottom layers. These are the characteristic features of layered architecture style. In our case, Card, effect data subsystem is the bottom layer of the architecture. This subsystem holds information about cards, effects and characters. As it is the bottom layer, this subsystem do not rely on any other layers. Also, this layer is the data source of the game entities system. Second layer is game and board subsystem. This subsystem consist of game, player and board information. All these objects needs information of cards and characters so this subsystem depends on card-effect data subsystem. Top layer is connector subsystem. It connects game entities with controller. It is dependent to other subsystems. Therefore, layered architecture fits our system.

5.4. Hardware/Software Mapping

Element Wars is a standalone application, which will not require any kind of web or network system to operate. Element Wars will be implemented in Java programming language, therefore we will use JDK (1.7). In order to run Element Wars, any standard desktop pc's configuration will be enough.

As hardware configuration, Element Wars needs a basic keyboard (for giving a name to a deck) and mouse for users to give input to the system. Since we will implement the project in Java, any standard desktop pc's configuration which has an operating system and JVM (Java Virtual Machine) is enough. In addition, we will benefit from java's platform independency.

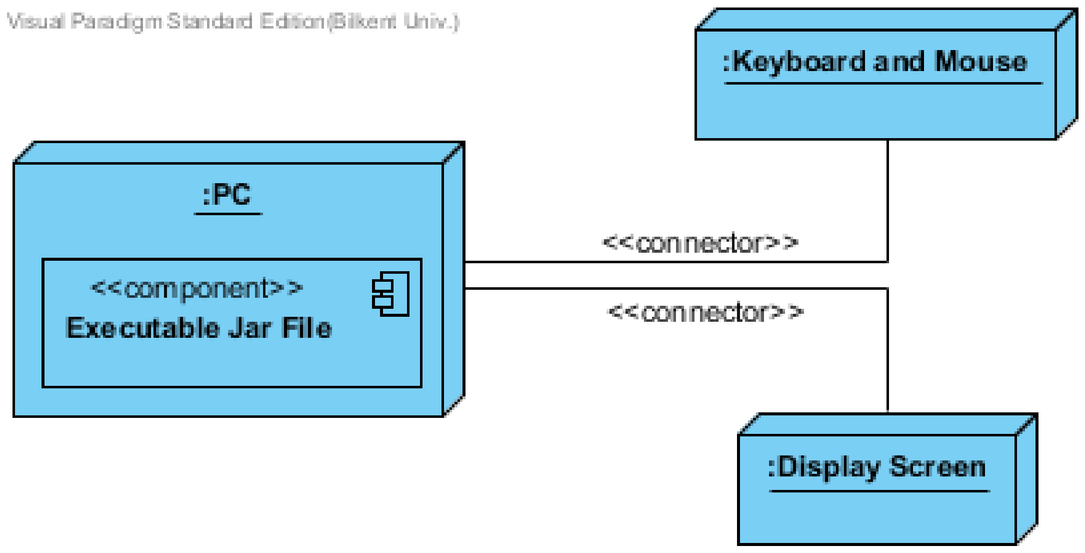


Figure 6 - Deployment Diagram

As deployment diagram shows, the system does not rely on extra hardware parts, just keyboard, mouse and a screen that can display the graphical outcome of the program are enough. In the diagram, Executable Jar File component may be divided into User Interface, Controller and Entity components but since they are shown in Subsystem Decomposition section, they are packaged as Executable Jar File for simplicity.

5.5. Persistent Data Management

There are two main persistent data types in our system. One of them is card information. For example, a player chooses a card from his hand to play, that card has some information like name, attack power, defense power, artwork image and so on. All these properties should be stored somewhere. Second type of data is deck information. When a player constructs a deck from available cards, that deck information should be stored as well so that player may play with that deck several times.

We use filesystem to store this information. We do not prefer a database system because our data is written by single writer and data we would like to store is not so complicated. Card information is stored in project path as .cd text files. Deck information is stored as .deck files.

5.6. Access Control and Security

Since there is no user specific information, system does not need an access control. Every time program is opened, existing decks will be available and any time a deck is deleted, it will be gone forever, in other words it will delete the existence of that deck from program. Since game is available for everyone using that computer or having the files required is free to use decks and edit decks there is no security issue. Because there is no information to be protected

5.7. Global Software Control

In our program, we decided to use event-driven control flow mechanism. Since it supports object oriented programming and our system's flow is controlled by events, event-driven mechanism fits perfect. Consistently with its Model-View-Controller architecture, program waits for an event to occur, which is a mouse click or keyboard typing in our case, updates the views. Thus, when an event occurs, detection will be made by a part and handling will be made by another part. User Interface subsystem gets the input from the user and delivers it to the Controller subsystem. Controller checks the kind of the event and does make that change in the views related with the event. The control is more decentralized because different objects decide on the actions by evaluating different events. Decentralized design means distribution of dynamic behavior to objects. There is no main controller that decides everything.

5.8. Boundary Conditions

5.8.1. Initialization

Starting the game is done by opening a jar file. No setup is required. In order to open the game, user must have java runtime environment installed on computer and corresponding jar file must be double clicked. Which starts up the game and enables user to enjoy the game.

5.8.2. Termination

Exiting the game is done by clicking "exit" button in main menu. The program will run in java and open as a new page. Since switch between pages requires more effort in order to terminate the java program. We decided to add exit button and that will terminate the program.

5.8.3.Exception

Exceptions may occur if text files of the game is changed manually, which corrupts the data in it. The solution for that is: if program can not open a deck, it will open a default deck which is in the program itself. Therefore, only loss is in the deck not in the program and since cards are still there new deck can be build with same cards.

6.Object Design

6.1.Design Patterns

6.1.1. Facade Pattern

Facade pattern is a structural software engineering design pattern where an object that provides a unified interface to a set of objects in a subsystem. It separates a subsystem from other subsystems or classes which causes a closed architecture. As a result we want to accomplish low coupling.

Subsystems have connector classes which take role of interfaces for other subsystems. We have two interface classes for this purpose. ModelConnector class contains every functionality of model classes that connector subsystems needs. Connector classes do not communicate entity classes of the system, but they communicate entity classes by using this interface class called ModelConnector. The other interface class for view subsystem is ControllerConnector. It does the same thing, creates a library of controller classes in order to be used in other subsystems. View controller use ControllerConnector to manipulate model classes. In other words, these classes are like application programming interface (API) for other classes.

We chose model-view-controller architecture style so we have concrete subsystem divisions. Therefore, we think that facade design pattern suits very well to our system because these subsystem should communicate with each other somehow. If all of them communicate with each other without any control mechanism, it causes complexity that we cannot handle. That is why we chose facade design pattern.

6.1.2. Observer Pattern

Between controller - model and between view - controller subsystem facade pattern is used. However for view - model connection, we used observer pattern. For facade pattern a class in a subsystem provides an interface for other subsystem. In our case, that interface would be in view and model classes use this interface to connect view classes if we consider facade pattern. However it creates a big problem in terms of encapsulation. Model classes should be independant and more stable. If we add an interface for every view, it makes model unstable and dependant to these interfaces. At this point, we give up from facade pattern and decide on observer pattern.

Observer pattern is used for observing the changes in a model object's state. It doesn't know anything except the changes in the state of objects it's tracking. In our project Element Wars, we use Observer pattern to show the state of the model classes to the user interface. It keeps track of the changes which can be many different possibilities and shows them on the screen. For example, let's consider a new turn scenario. Player draws a card from his deck and add that card to his hand. After that, there are changes in model classes' states. Number of card in deck is decreased and a new card is added to hand. All these state changes trigger models to notify their observers and then observers (user interface) execute their update methods and the change can be observed in the user interface.

6.1.3. Interpreter Pattern

As indicated in requirement analysis, our game contains many cards. One bad way to implement these cards is to hardcode all of them. With this plan, we may be nominated for worst software developer award. However we chose a design pattern to implement these card. We created .card files in disk. These files contain information about properties of different cards in a format (Json) and some classes in the system are responsible to create card objects using these files. This design pattern makes future changes so easy. For example, in order to add a new card into the game, the only thing we should do is to create a .card file in project folder. We do not even need to rebuild the project.

For card effects, this pattern is almost an obligation. There might be very different card effects (For example: deal 2 damage to a character, destroy a character, swap two character) and we cannot hardcode every possible card effect. With this pattern, interpreter class can interpret effects and create proper objects or actions. Note that, card effects are not considered as a part of the project right now. We may add this feature in future versions.

6.2.Class Interfaces

<u>Controllers:</u>

1. ActionManager.java
2. BoardManager.java
3. CombatManager.java
4. ControllerConnector.java
5. GameManager.java
6. Manager.java
7. Tester.java
8. TurnManager.java

<u>Models:</u>

1. Account.java

<u>Properties:</u>

Sting name:

Deck deck:

String aiName:

<u>Constructor:</u>

public Account():(exception)

public Account(String name):

public Account(Deck deck):

public Player createPlayer(Side side):

<u>Methods:</u>

public String getName() :

public String getDeck():

public String getAIName():

public String setName() :

public String setDeck() :

public String getAIName():

2. Board.java

<u>Extends:</u>

Observable:

<u>Properties:</u>

int Size:

Player WhitePlayer:

Player BlackPlayer:

ArrayList<Character> whiteSide:

ArrayList<Character> blackSide;

<u>Constructor:</u>

public Board(Player whitePlayer, Player blackPlayer):

<u>Methods:</u>

public boolean putCharacter(Character character, int position, Side side):

public boolean putCharacter(CharacterCard card, int position, Side side):

public boolean removeCharacter(int position, Side side):

public void prepareForBattle(Side side):

public boolean swapCharacters(int pos1, int pos2, Side side):

public void killCharacter(int position, Side side):

public Character getCharacter(int position, Side side):

public ArrayList<Character> getSide(Side side):

public Player getWhitePlayer():

public void setWhitePlayer(Player whitePlayer):

public Player getBlackPlayer():

public void setBlackPlayer(Player blackPlayer):

3. Card.java

Extends:

Observable:

Properties:

int id

String name:

EnergySet cost:

Effect effect:

Constructor:

public Card(int id, String name, EnergySet cost, Effect effect):

public Card(int id, String name, EnergySet cost):

public Card copy():

Methods:

public int getId():

public void setId(int id):

public String getName() :

public void setName(String name) :

public EnergySet getCost():

public void setCost(EnergySet cost):

public Effect getEffect() :

public void setEffect(Effect effect) :

4. CardImporter.java
5. CardPile.java
6. Character.java
7. CharacterCard.java
8. CorruptedFileException.java
9. Deck.java
10. DeckIO.java
11. Effect.java
12. Energy.java
13. EnergyPalette.java
14. EnergySet.java
15. Game.java
16. Hand.java
17. ModelConnector.java
18. Player.java
19. Setting.java
20. SpellCard.java

7.Conclusion

In this report, we tried to explain what is our project, what it does, which functionalities it has, what we aim to do, how we will do design it and further details about the implementation of the project. According to our analysis report we tried to decide the functionality and specifications of our game. By looking at use case and activity diagrams we choose how to design our game by the design raport we decided how to implement our game. We hope you will like it.