Bilkent University
CS 319

Element Wars Project

Design Report
3 December 2014

Group 8 Members
Selçuk Gülcan
Umut Hiçyılmaz
Serdar Demirkol

Table of Content

1. Introduction

We are group 8, we proposed a card game called Element Wars in our analysis report. To remind, Element Wars is a turn based single player collectible card game. The game is played against an AI. Player constructs a deck from a collection of cards. Player uses these card to defeat opponent by reducing its life point to zero.

2. 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.

- **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.

- **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.

- **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.

- **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.

- **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.

- **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.

3. 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.

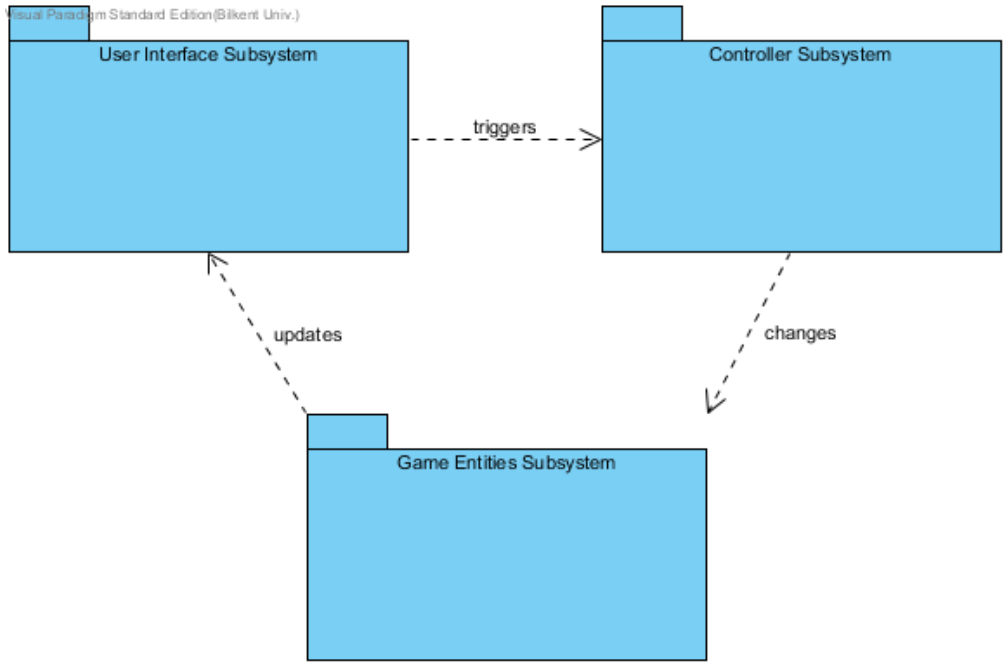

Figure 1 - Subsystems

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.

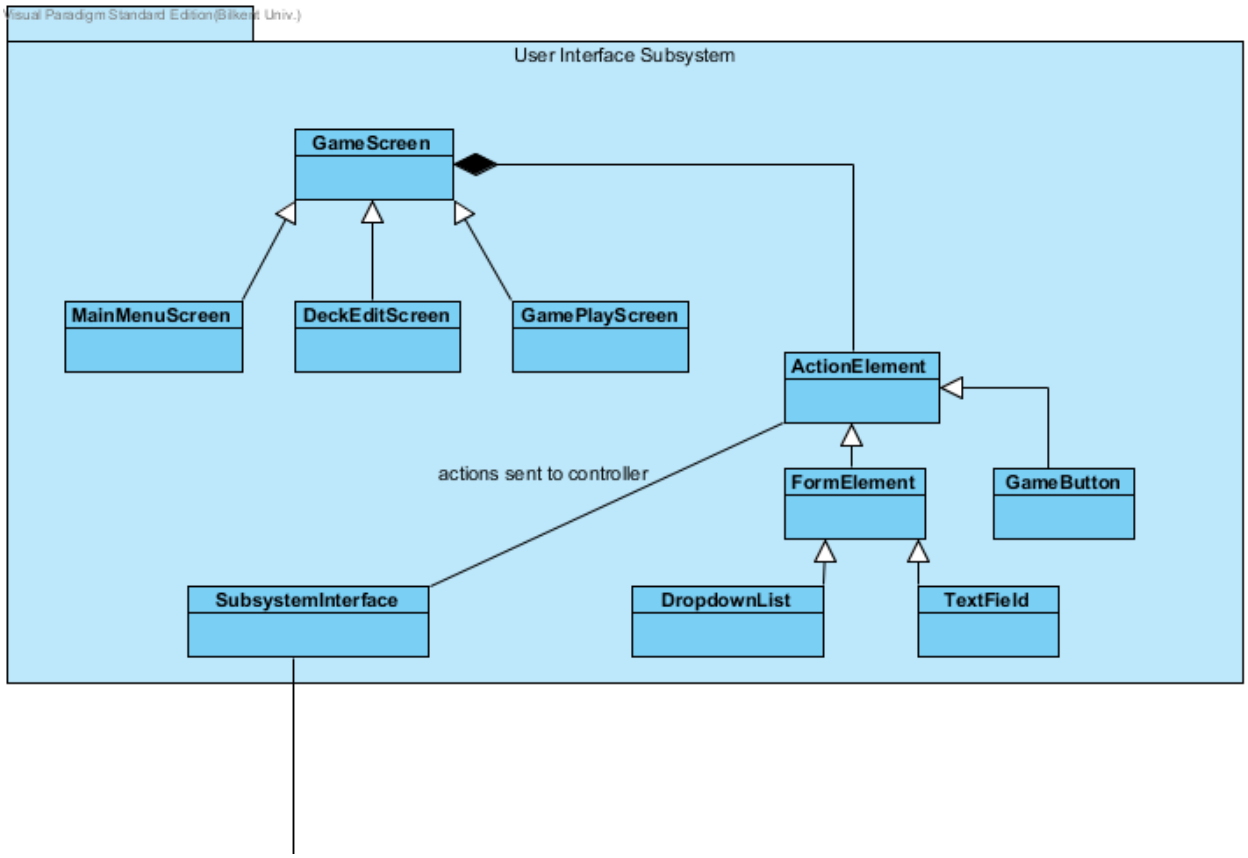

Figure 2 - User Interface Subsystem

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.

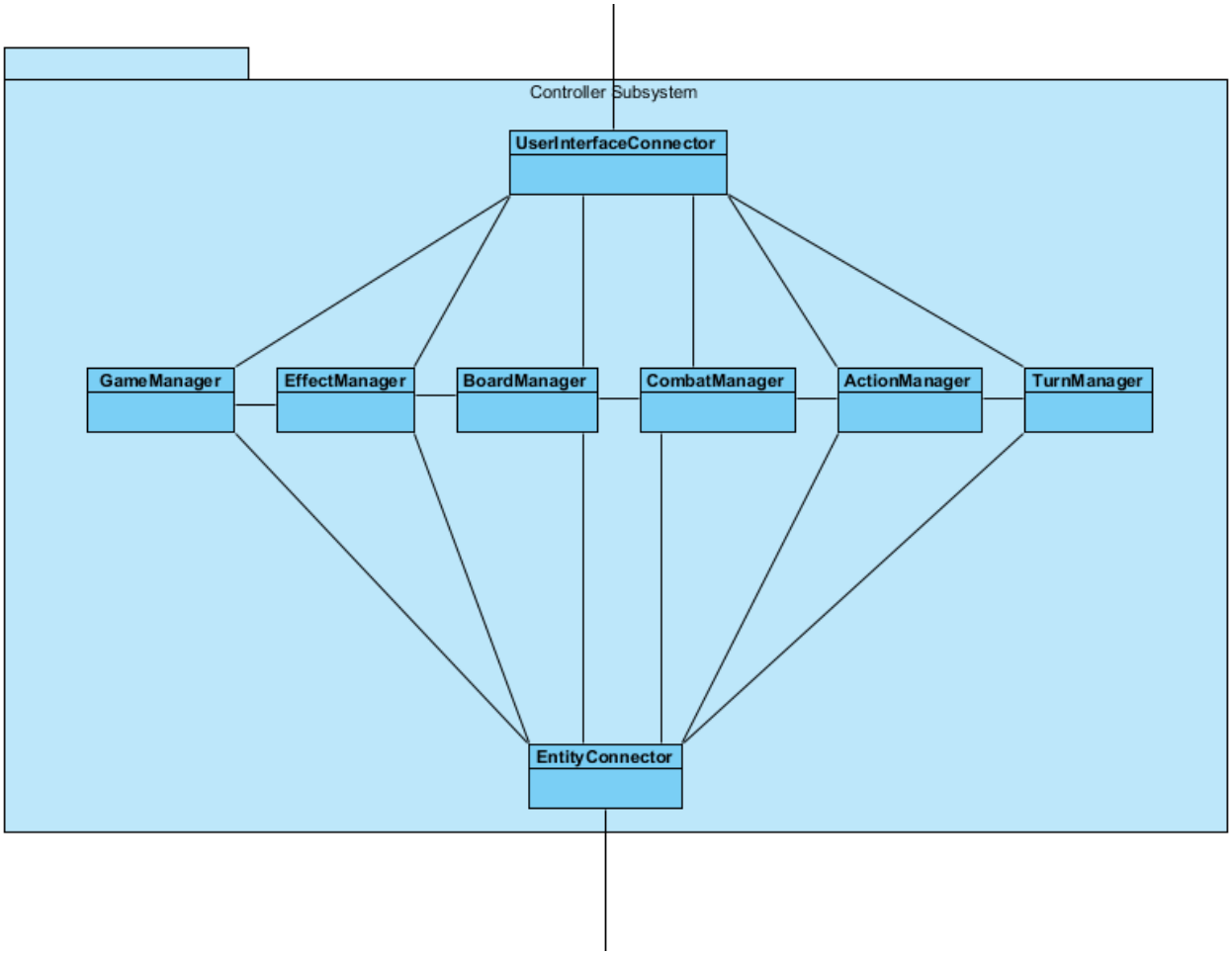

Figure 3 - Controller Subsystem

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.

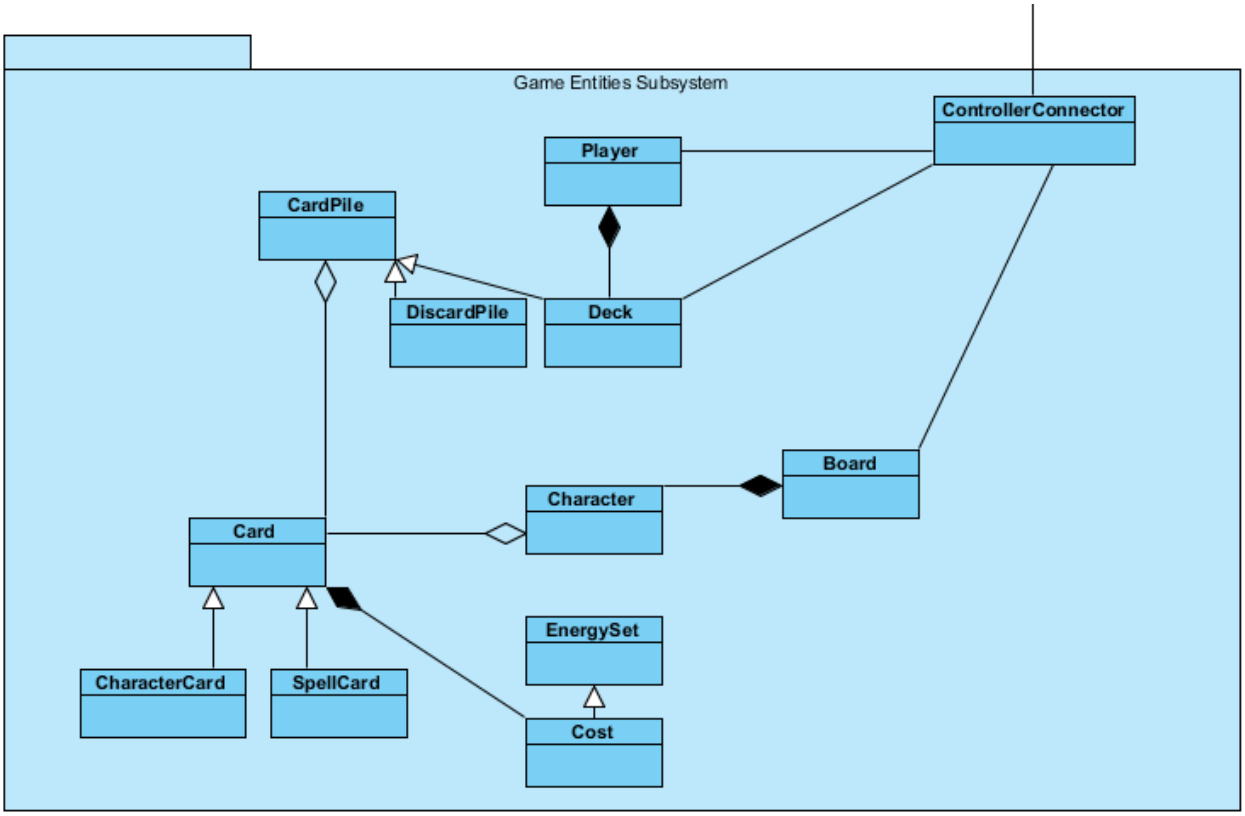

Figure 4 - Game Entities Subsystem

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.

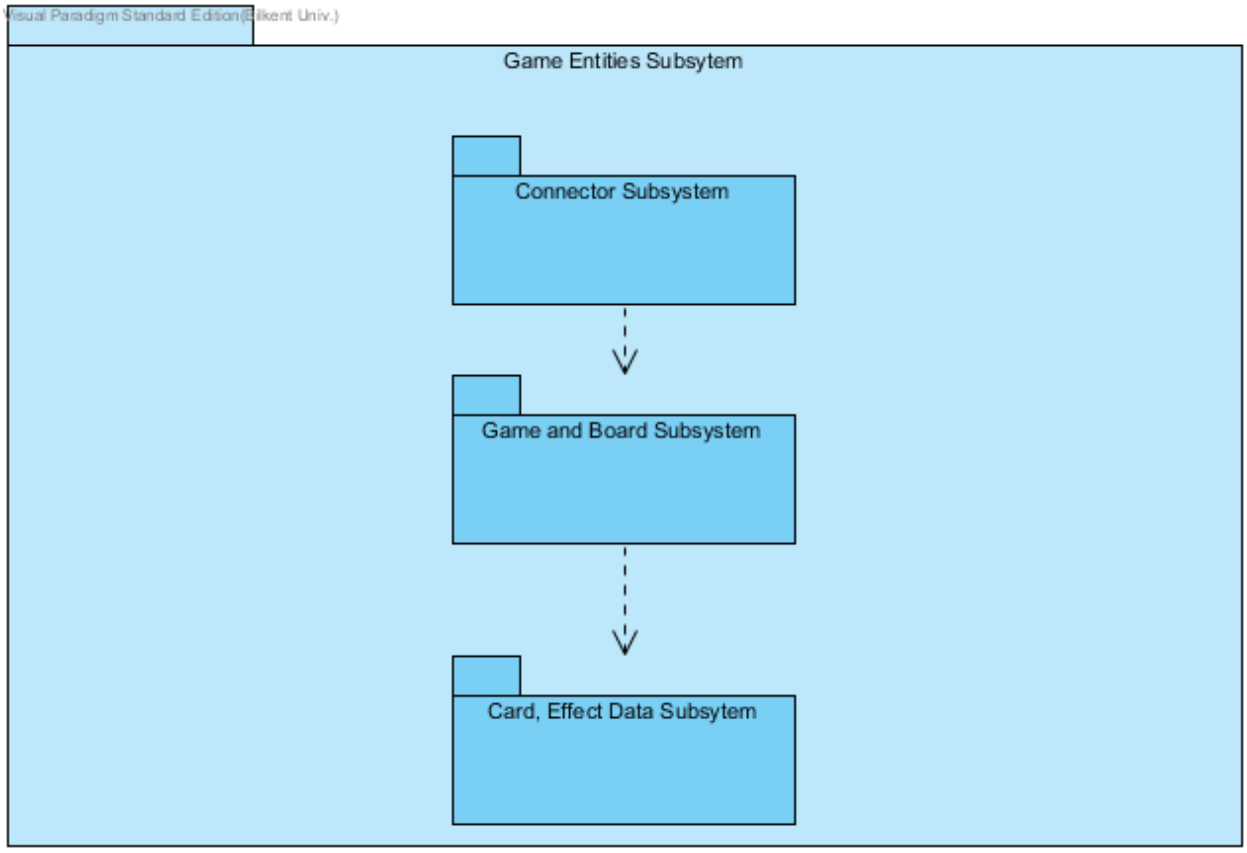

Figure 5 - Game Entities Subsystem Decomposition

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

4. Architectural Styles

We use two different architectural styles: Model-view-controller architecture (MVC) and layered architecture. The reason why we choose these architectural styles can be understood by looking at our subsystem decomposition figures.

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.

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. 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.

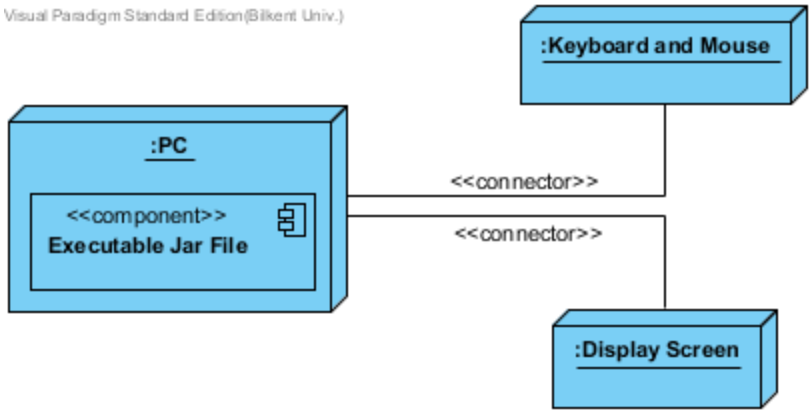

Figure 6 - Deployment Diagram

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.

6. Addressing Key Concerns

6.1. 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.

6.2. 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.

6.3. 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.

6.4. Boundary Conditions

This part is about start and end of the program. How they are done and through what the game will be opened? These questions will be explained in detail in the upcoming sections.

- **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

- **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.

- **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.

6.5. Management Intractability

When a group of people works on the same project, many problems may arise. For example, the work of a developer may depend on the work of another developer so the developer have to wait the other developer to finish his job or he have to complete his part without knowing the other part. Another problem is that in weakly organized projects, when a developer changes a part of the system, other parts do not work anymore. As the number of people involved in the project increase, this kind of problems increases proportionally. We want to minimize these problems as much as we can. To do so, project work should be divided into pieces that do not have high dependencies. There should be some interfaces between these pieces and all communication should be done via these interfaces. We believe this will decrease the management problem. When something is changed, other pieces are not affected as long as the interface between these pieces works smoothly.

Also, our architecture style choice make our project easy to manage. With model-view-controller architecture style, the project may be divided easily with low dependency. By doing so, we believe a group of people can work on the project without facing such difficulties mentioned above.