Bilkent University

Department of Computer Engineering

Object Oriented Software Engineering Project

CS-319/1: Hearthstone

Final Report

K. Bartu KÖMÜRCÜ ,Melike KESKİN, Mert CEYLAN

Course Instructor: Ertuğrul Kartal TABAK

Progress March 23, 2015This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Object Oriented Software Engineering course CS319/1.

Contents

1. Introduction:

As the members of group#7 in cs-319 we are assigned to make an object-oriented software program and we decided to make a card game as our project. Our project is inspired on Blizzard's card game called Hearthstone: Heroes of Warcraft. The game will consist of a variety of heroes (or classes), and these heroes will have their unique abilities and certain set of cards which are only available to those heroes only. Also, there will a set of cards which are called "Neutral cards" which will be available to all "heroes".

In this report we will detailed information about the game and explain how the game works, with functional, non-functional requirements, scenarios and UML diagrams and models.

1.1 Why Hearthstone?

We chose this game as our topic because we all enjoy the game as the members of our group ,secondly games are always good examples of object-oriented programming and Hearthstone is one of the best game that demonstrates the object oriented programming principles because the game consist of lots of classes and there are tons of relationships between classes.

2. General Information about Hearthstone

Hearthstone is a card game which has 9 different heroes (or classes), which are warrior, shaman, rouge, paladin, hunter, druid, warlock, mage and priest. All of these classes have their special abilities and also special cards. Before starting a game the player should select a "hero" and make a deck with that hero before starting a game. There are 3 kinds of cards in the game, spell cards, minion cards and weapon cards. Minion cards mana cost, heath points and damage points and some of these cards have special effects. Once a minion played on the board they stay at the board until their health point is 0 or below. Spell cards have mana costs only and once they are played they can perform their effect and that’s it you can’t play the same card again. Weapon cards also have a mana cost, damage points and a durability point once they are played your hero is able to attack with that weapon and every time they something with that weapon the weapons durability decrease by 1 until it 0 or another weapon is equipped by that hero. After the user finished making the deck he will be able to play with that deck and that deck will be saved for future use. When the player starts a game there will be an opposing “hero” to play with and also there will be turns. The game randomly choses the first player. First player receives 3 random cards from his/her deck and the second player starts with 4 random cards from his deck. In this state of the game players are able to choose between those 3 or 4 cards. They can either keep all the cards or return some of them to their deck and draw new cards. When the card selecting phase finishes it’s the turn of the first player and every turn players draw 1 card from their deck and the player receives extra 1 mana added to the maximum mana pool until the the maximum mana reaches to 10. Also, in every turns beginning the players mana is restored to the maximum value and the last turns unused mana is not important. For example, in the beginning of the 2nd turn for the 1st player, 1st players mana is restored to 1 and receives a bonus maximum mana so the result is 2 available mana for that turn and the result will be always 2 mana even if the first player did not use his mana in the previous turn, and players have a resource called mana which can be used for playing cards or using hero abilities, the second player receives a card called “The coin” which can be used for 1 extra mana for 1 turn. Every players hero starts with 30 life points and the goal of the game is to reduce the opposing heroes life points to 0 using their cards and hero abilities.

2.1 Heroes and Their Abilities

All heroes have special ability called “hero ability” and these abilities all cost 2 mana and all hero abilities are unique to that hero.

2.1.1 Warrior

Warriors hero ability is to give himself 2 armor.

2.1.2 Shaman

Shamans hero ability is to summon a totem. There is 4 different kinds of totems. The totems are “searing totem”, “healing totem”, “stoneclaw totem”, and “wrath of air totem”

2.1.3 Rogue

Rogues hero ability is to give herself a dagger

2.1.4 Paladin

Paladins hero ability is to summon a silverhand recruit

2.1.5 Hunter

Hunters hero ability is to deal 2 damage to the opposing hero

2.1.6 Druid

Druids hero ability is to give himself 1 damage for one turn and gain 1 armor

2.1.7 Warlock

Warlocks hero ability is to deal 2 damage to himself and draw 1 card

2.1.8 Mage

Mages hero ability is to deal 1 damage to anyone

2.1.9 Priest

Priests hero ability is to heal anyone by 2 health points

3. Proposed System

3.1 Functional Requirements

3.1.1 Play

Play option should take first place in main menu interface. The game should present play button when users start the game. Then users should be able to pass Choose Deck option. The game cannot be played without constructing any decks

3.1.2 Choose Deck

Both users should choose a deck which were constructed before.

3.1.3 Delete Deck

Users are able to delete existing decks

3.1.4 View Cards

Construct Deck should take place after the Play option in main menu interface. In Construct Deck option users should see their available cards. They can choose to construct a new deck.

3.1.5 Construct Deck

Construct Deck option should give opportunity to choose users to make their own deck which consist of 30 cards.

3.1.6 Concede

User should be able to forfeit from continuing game.

3.1.7 Help

Users should be able to read the game rules before starting a game.

3.2 Non-functional Requirements

3.2.1 User-Friendly Interface

Users should only play this game with mouse clicks. Interface should be self-explanatory for a user who knows the game rules (it is also provided in the help section). Visuals should be non-overlapping and clear.

3.2.2 Game Performance

All user interface object should response with max one second delay. The game will start in 3 seconds.

3.3 Pseudo Requirements

The game will be implemented in Java.

3.4 System Models

3.4.1 Scenarios

3.4.1.1 Scenario A

Mert opens up the game, chooses a deck and plays the game with his friend Melike, after the game ends by Melike conceding.

3.4.1.2 Scenario B

Bartu opens up the game, views the cards, decides to make a deck, he chooses his cards and creates his deck.

3.4.1.3 Scenario C

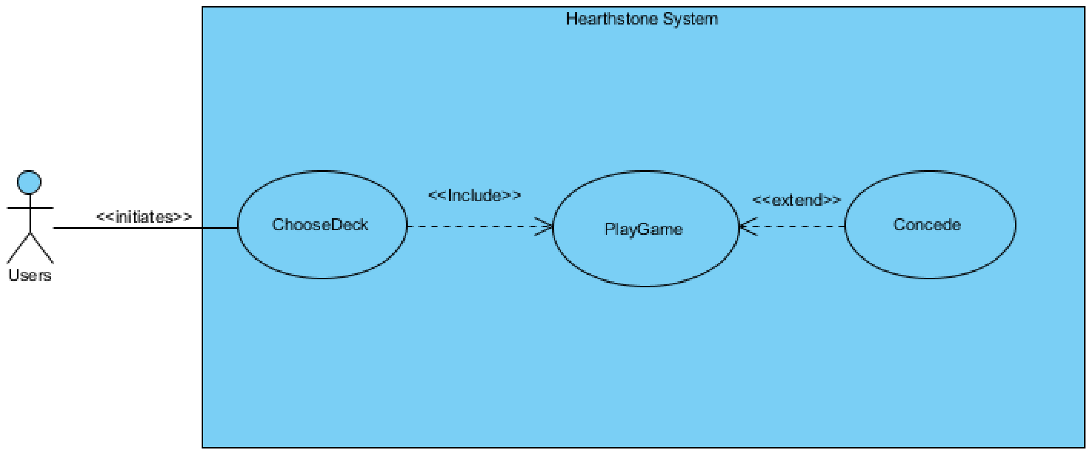
Mert opens up the game, creates a deck and plays the game with Melike, Mert chooses his deck and Melike chooses another deck, Melike wins the game.

3.4.1.4 Scenario D

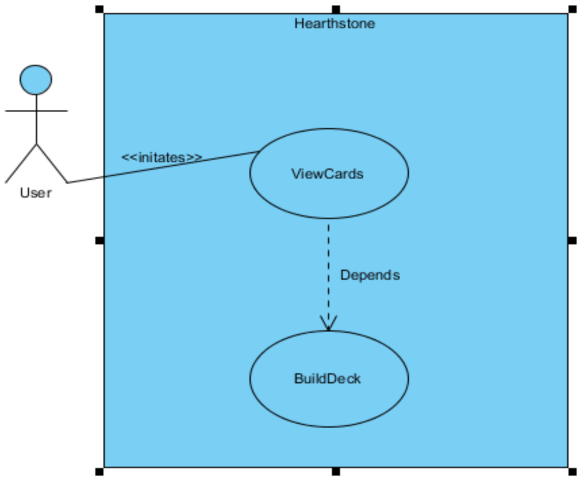
Mert opens up the game, tries to play a game, but he can't play the game because he has no deck.

3.4.2 Use-Case Diagrams

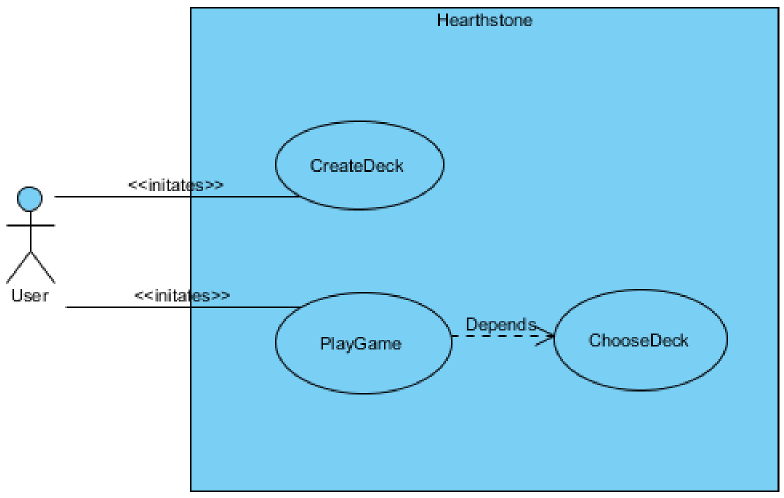
3.4.2.1 Use-Case for Scenario A



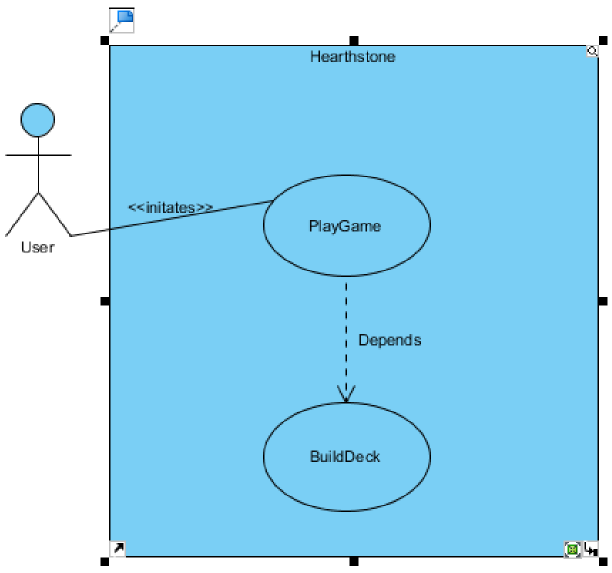
3.4.2.2 Use-Case for Scenario B



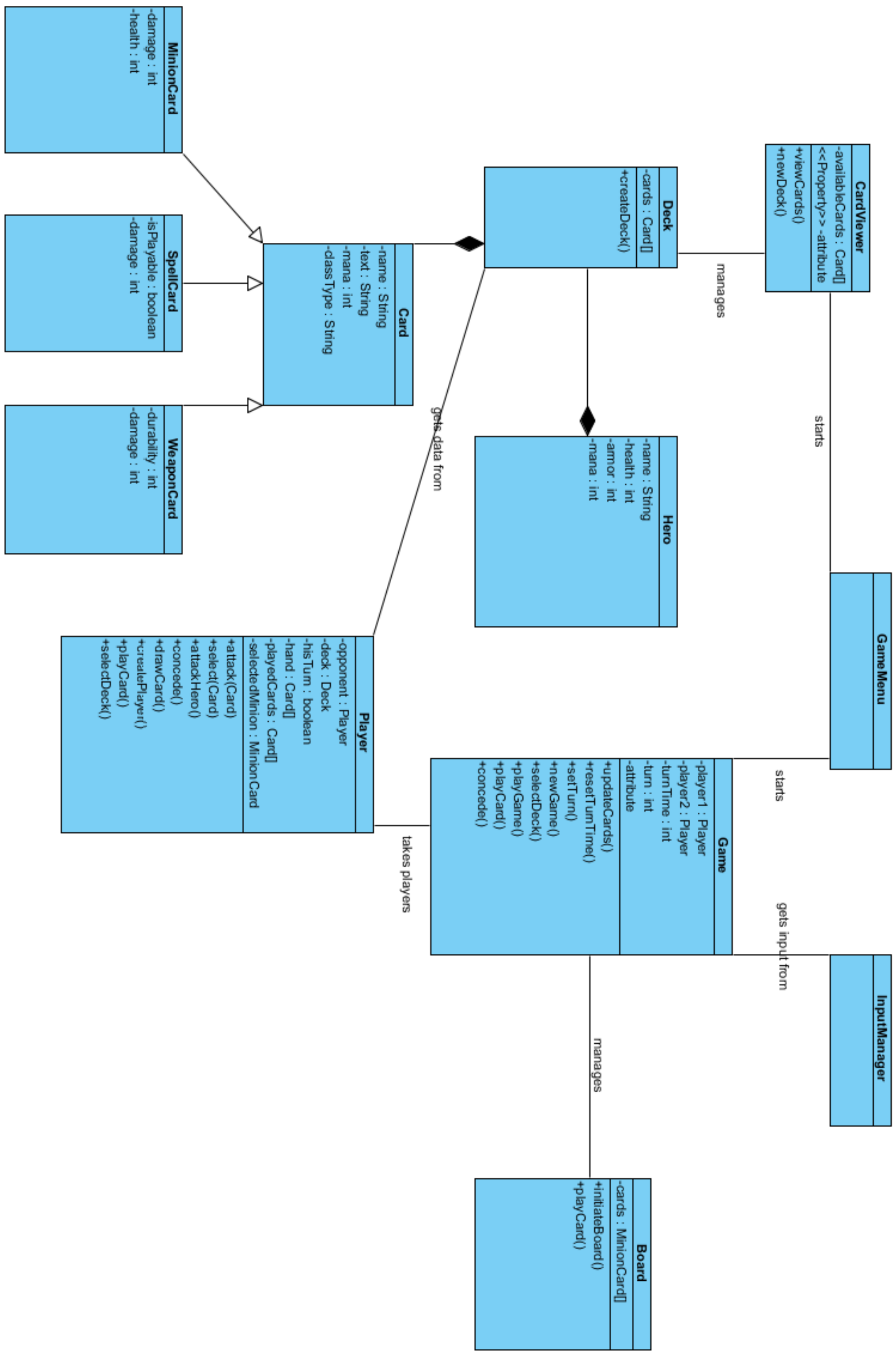
3.4.2.3 Use-Case for Scenario C



3.4.2.4 Use-Case for Scenario D

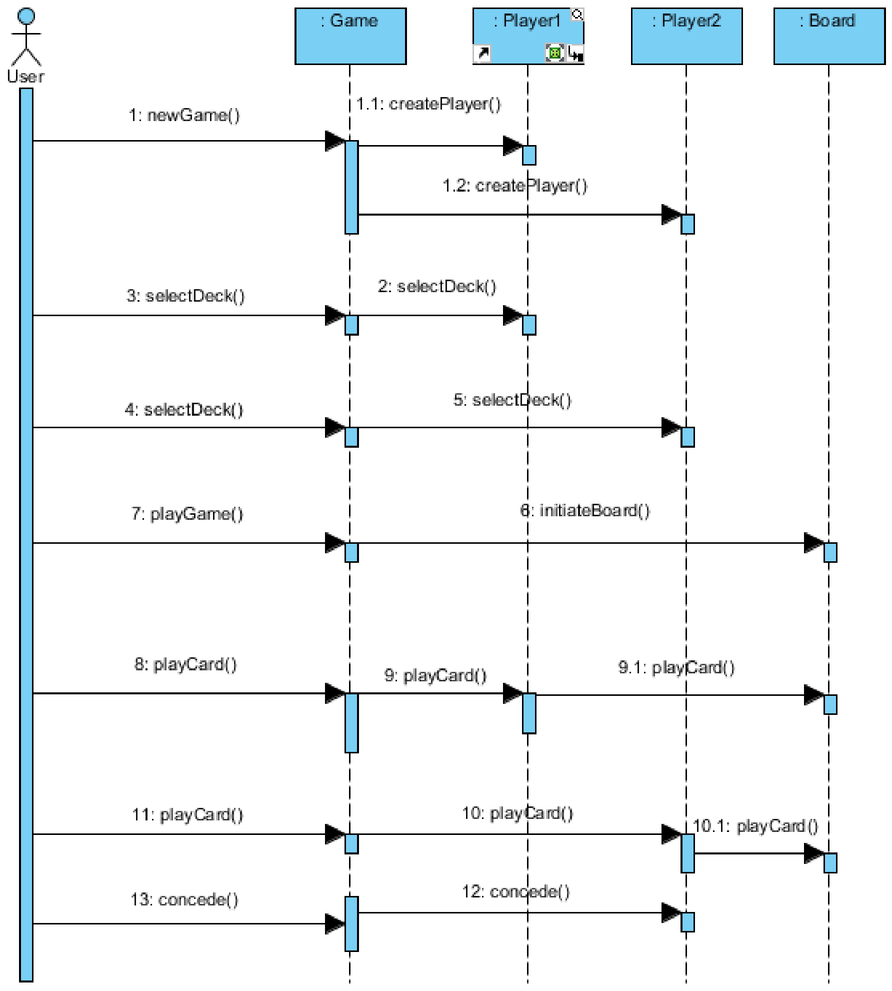


3.4.3 Object and Class model

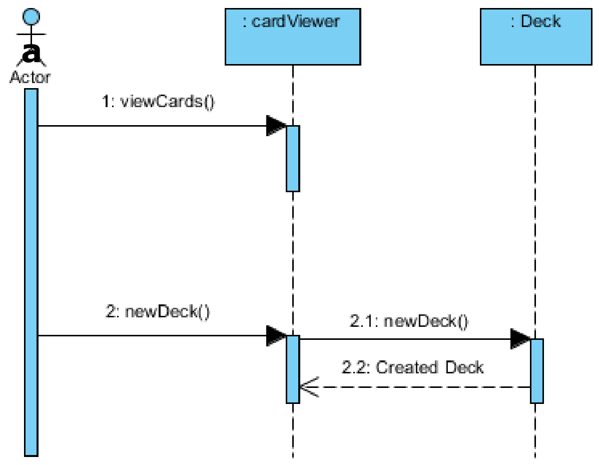


3.4.4 Dynamic Model

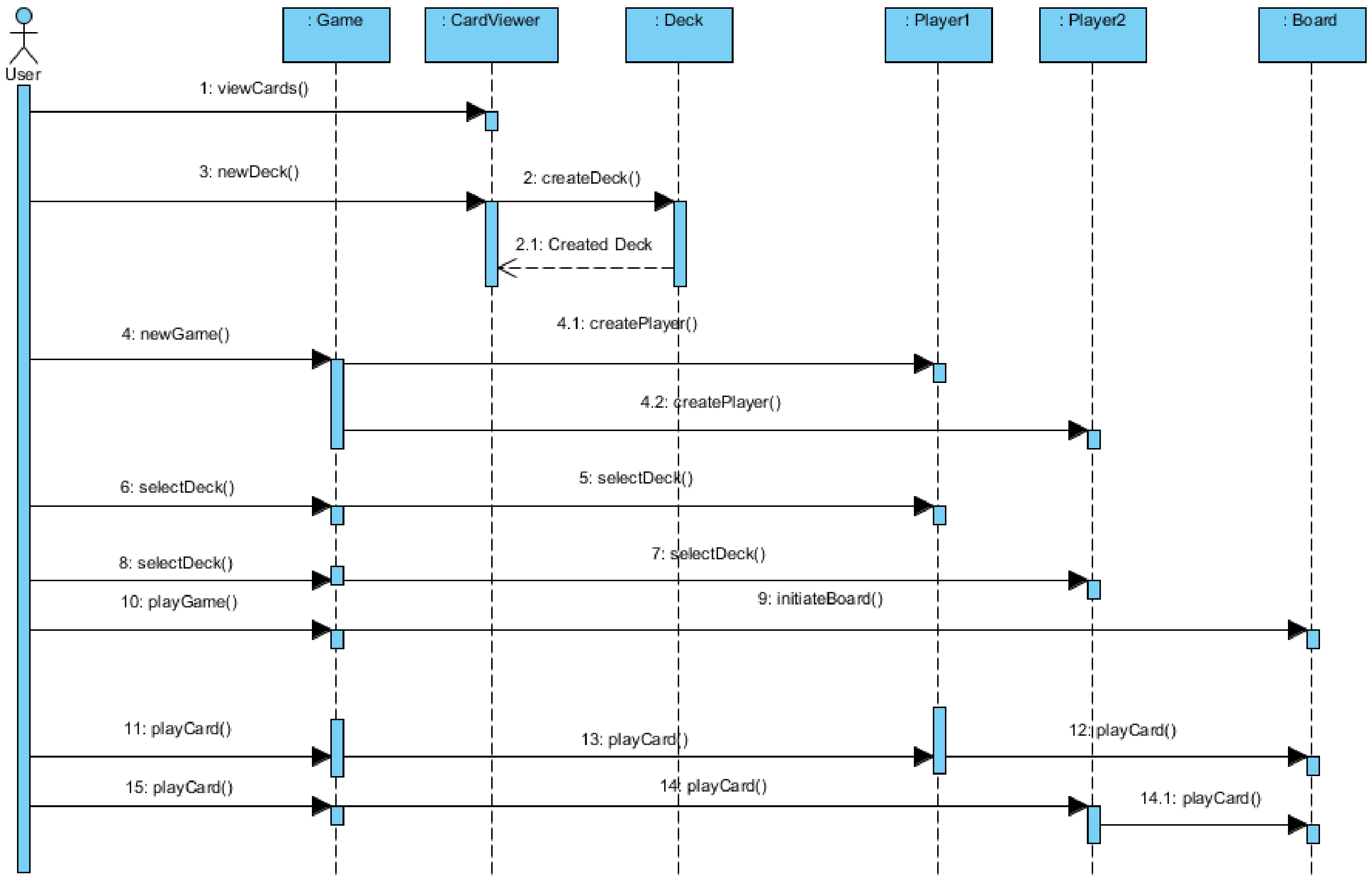
3.4.4.1 Sequence Diagram 1



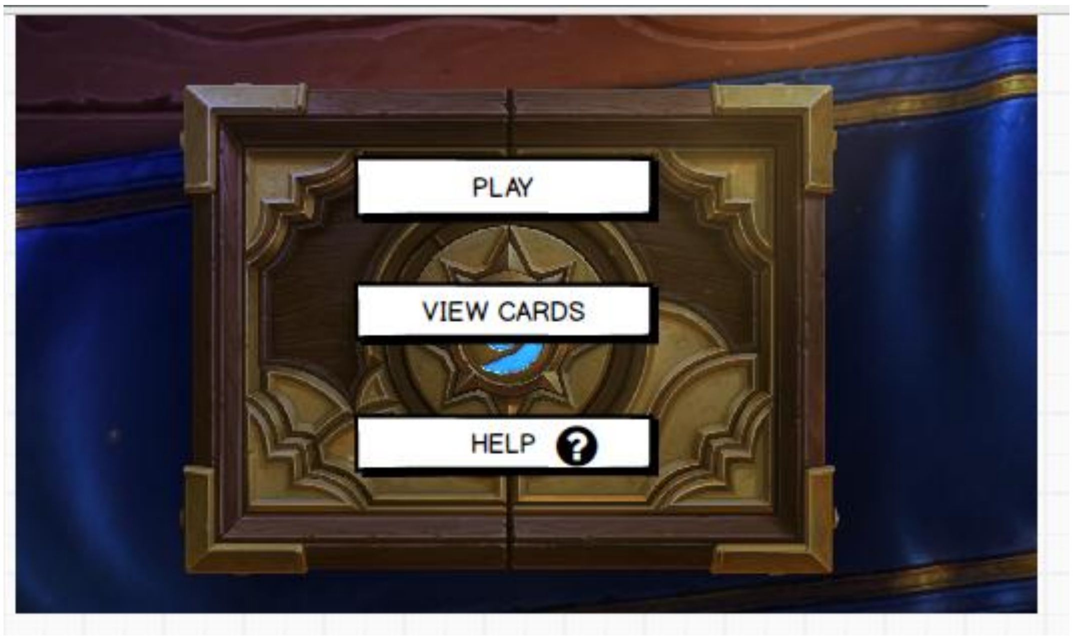
3.4.4.2 Sequence Diagram 2

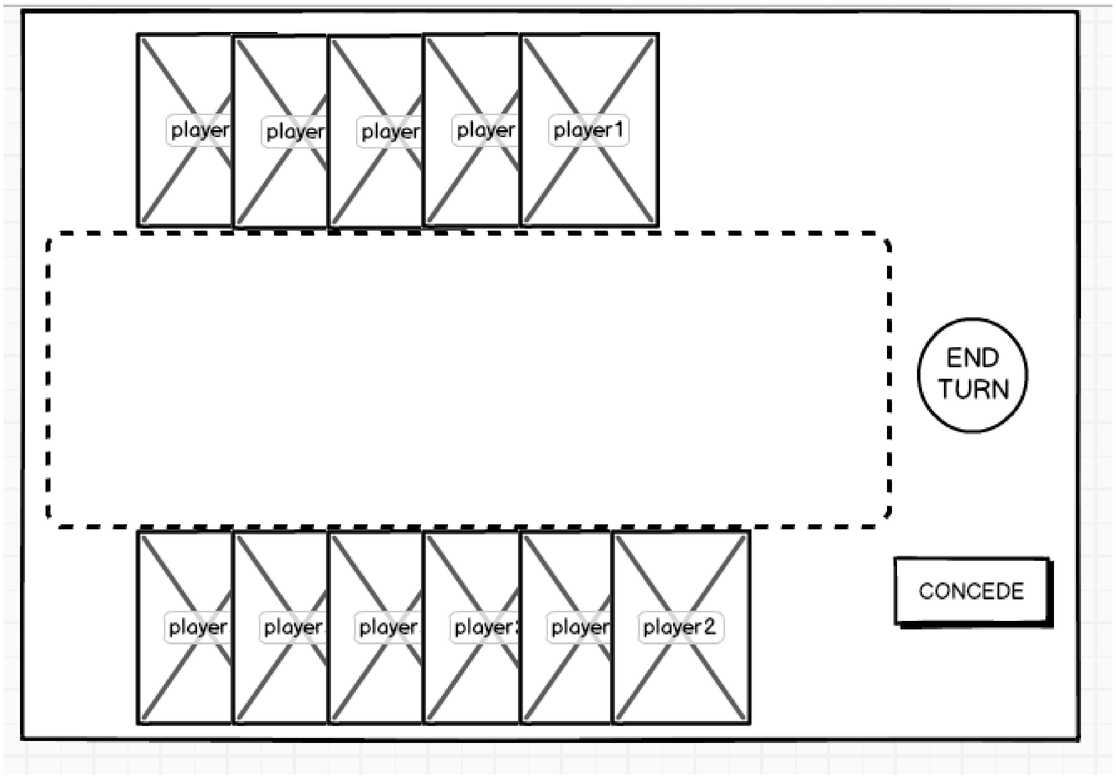


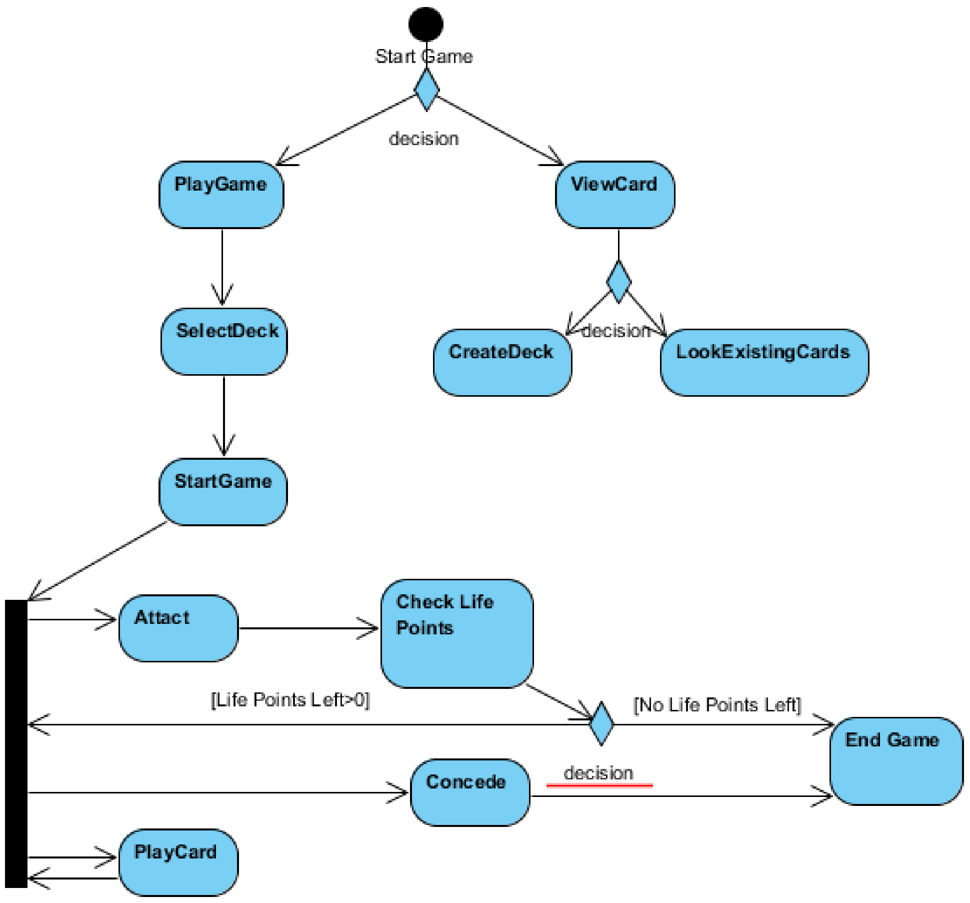
3.4.4.3 Sequence Diagram 3

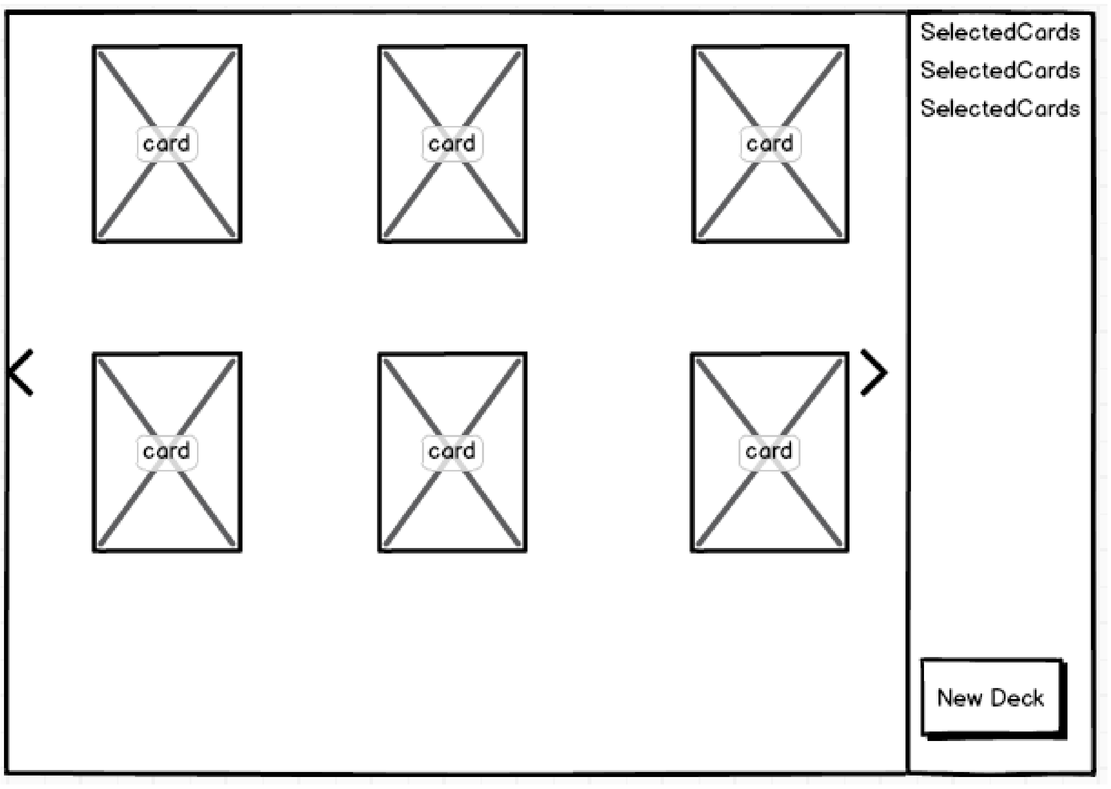


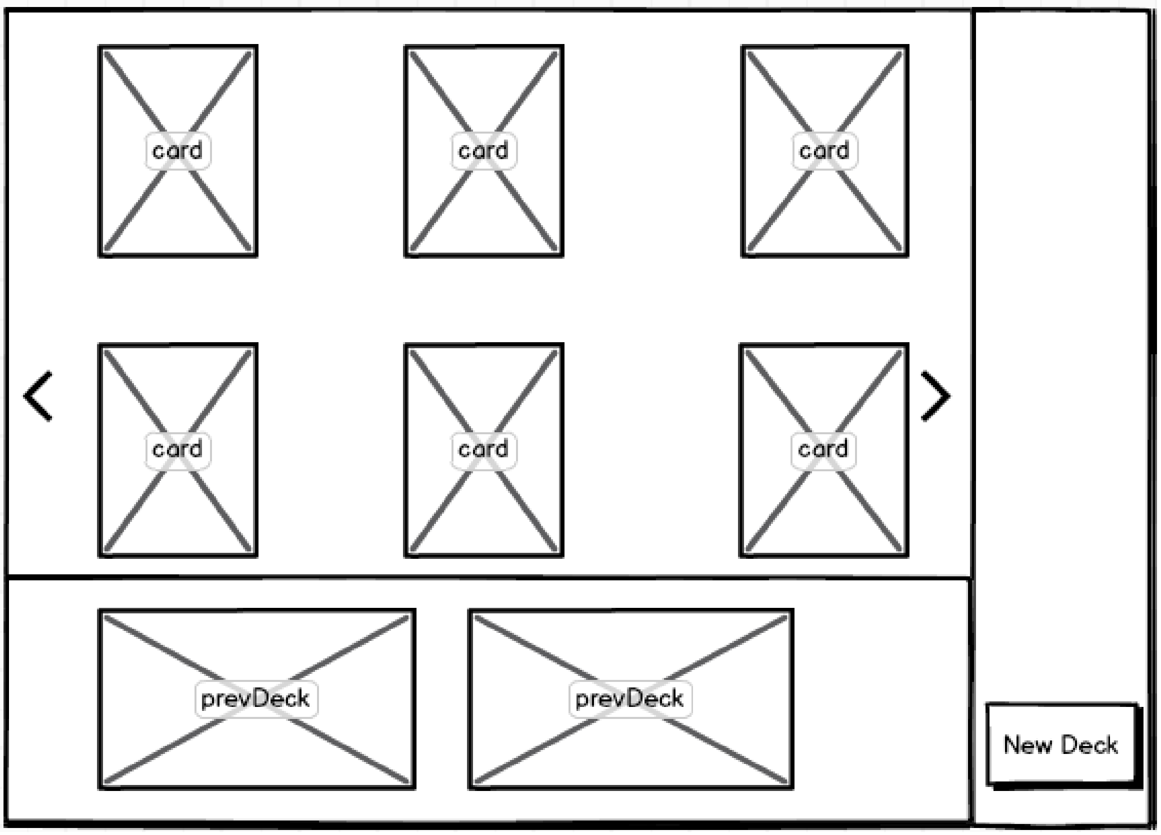
3.4.5 User Interface

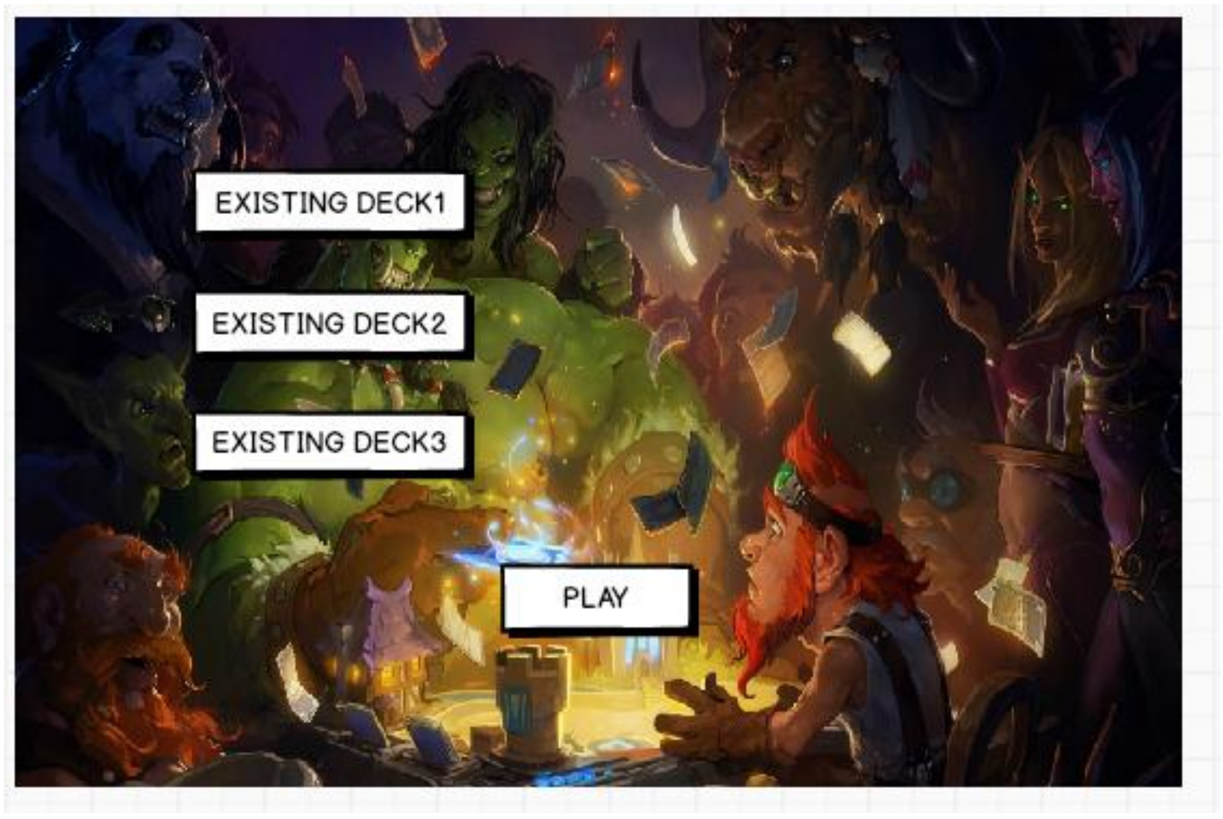





Start Game
decision
PlayGame
ViewCard
SelectDeck
decision
CreateDeck
LookExistingCards
StartGame
Attact
Check Life Points
[Life Points Left>0]
[No Life Points Left]
End Game
Concede
decision
PlayCard


card
card
card
card
card
card
SelectedCards
SelectedCards
SelectedCards
New Deck


card
card
card
card
card
card
prevDeck
prevDeck
New Deck


EXISTING DECK1
EXISTING DECK2
EXISTING DECK3
PLAY

5. References

6. Design

As RandomSoft our design is to make a 2-player card game which is very similar to Blizzard Entertainment's card game Hearthstone. It is a turn based card game where the goal is to kill your opponent by reducing their health points to zero by using a deck which consists of various minions and spells.

6.1 Design Goals

Our main design goal is to make the game as simple as possible, by using the techniques we learnt through out CS 319. Therefore our main goal will be to reduce coupling and increase the cohesion. Also we want to make the documentation of our project as clear and clean as possible for other people who want to use and improve our code.

6.1.1 Reliability

We want our game to run as smoothly as possible without any errors and crashes and this is our main goal. We want to prioritize on this because the most important thing for us is our game is working and doing its job properly, second reason is that if the game is buggy and contains lots of errors then our users won't want to play our game.

6.1.2 Modifiability

Since we are making a card game we want to add new content whenever we want to our program. So our second most important design goal is that our program could be modified easily and without too much effort because maybe in the future we may add some new cards.

6.1.3 Maintainability

Our code should be written in a generic way that when we want to change something we do not encounter some extra errors or new bugs and by nature our project is really open to adding new content. So our documentation and our code should be maintainable so that we can change and manage it easily in the future.

6.1.4 User-friendliness

By nature card games are easy to learn and hard to master, because of thousands of different combinations and also the players should know the properties of each card in our game to play it properly. The most important aspect of our game is the game can be learned easily. So our games interface must be easy to understand and self-explanatory. Also in game cards and their properties should be self-explanatory and easy to understand.

6.1.5 Reusability

Our code should be reusable because as we mentioned earlier, our game is really open to new content. When implementing these new contents, we should be able to use our former work would be really efficient time wise.

6.1.6 Portability

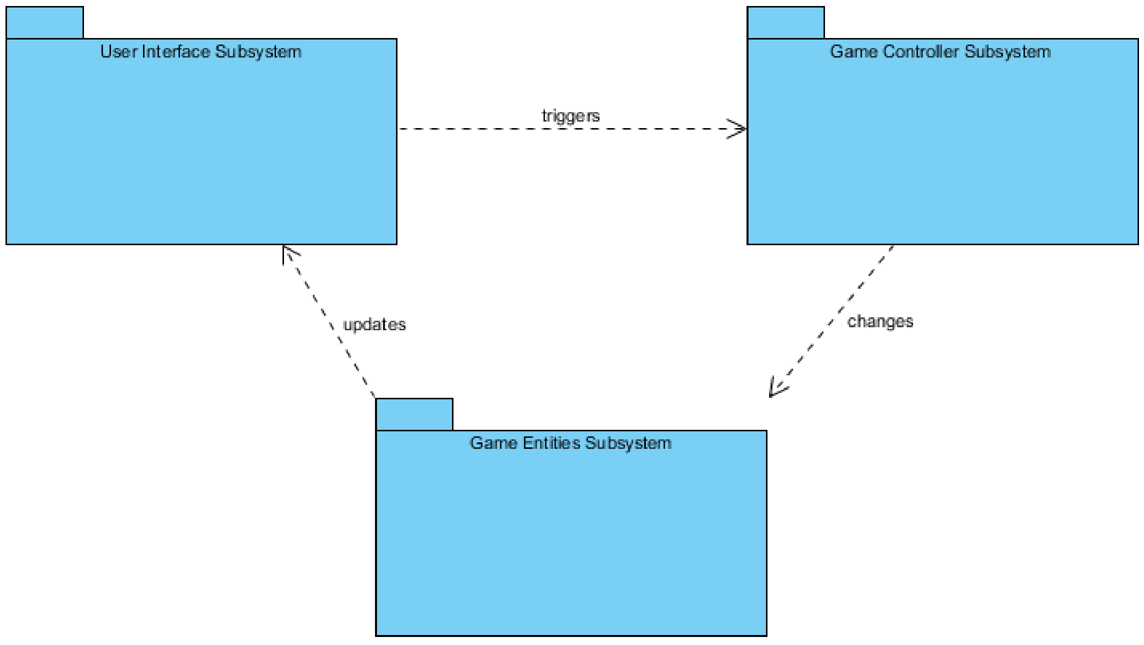
We are designing our game in java, so our game should run on both Windows and Linux without any compatibility problems. And also the decks that are created in our game should be saved in an external file. So if the player changes environments he should not have any problems about losing his decks that he previously made.

6.1.7 Robustness

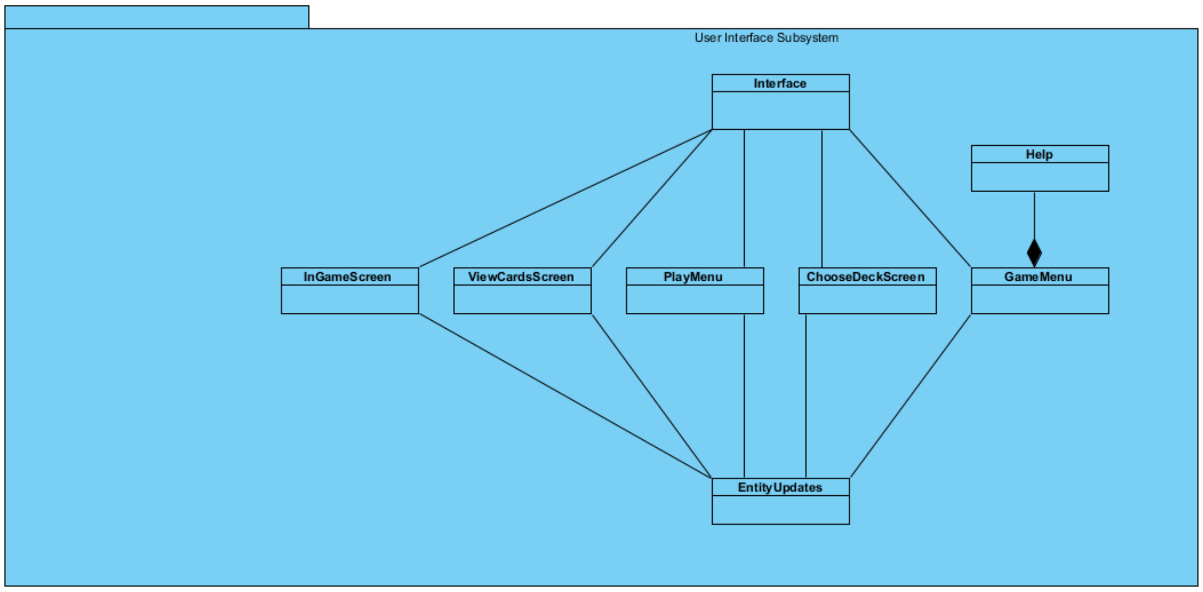
Our game has a lot of replay value. Small changes made on a deck may mean lots and lots of different possibilities in the gameplay. We do not want to have any kinds of errors while adding new cards to the game and also the decks inside the game.

6.2 Sub-System Decomposition

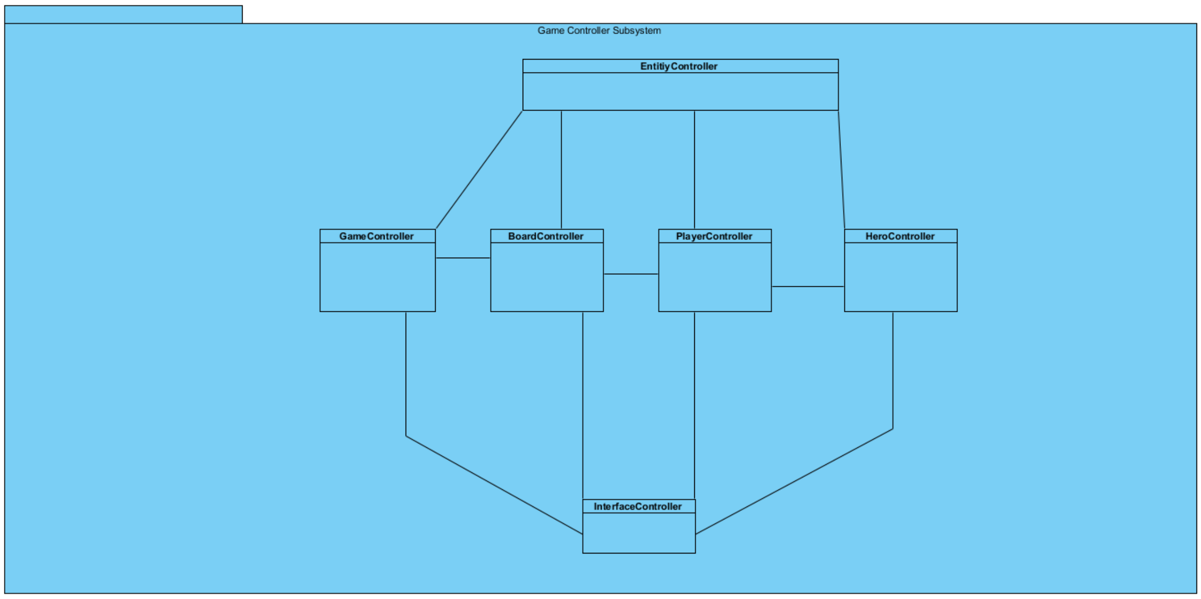
In this section, in order to reduce the complexity of the program, the system will be decomposed into subsystems. In subsystem decomposition, the main goal is allowing changes. Therefore, high coherence and low coupling should be maintained.
The system has three main subsystems; User Interface, Game Controller, and Game Entities. They have minimum relationships from each other because of the chosen subsystem decomposition style.



User only interacts with User Interface subsystem. This statement is crucial to minimize the complexity of the system. User Interface subsystem will notify the Game Controller



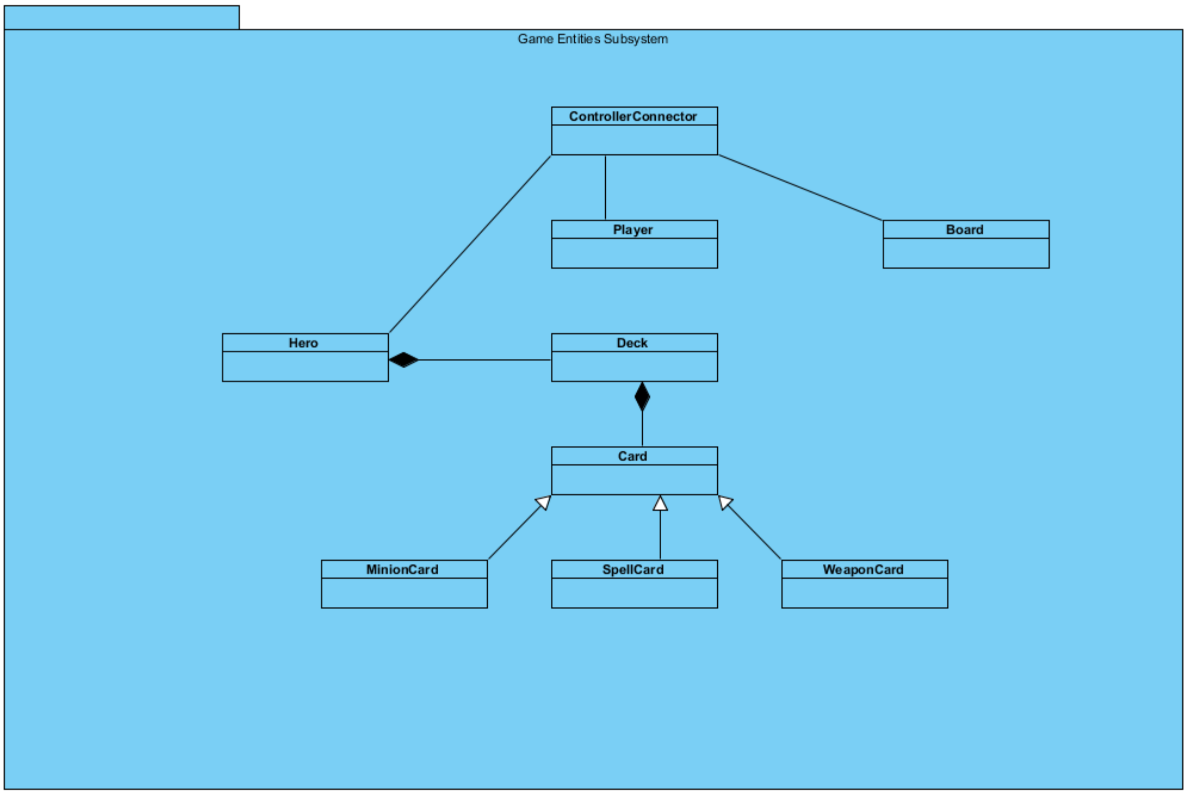
according to the User’s choices. Game Controller will check the eligibility of those choices, and calculate the outcome of the actions by communicating with Game Entities subsystem. Game Entities subsystem holds all of the data of the game entities. In this process, Game Controller might change the data in the Game Entities, and when the data is changed in Game Entities, the corresponding User Interface classes will be notified.
User Interface subsystem contains; GameMenu, InGameScreen, ViewCardsScreen, ChooseDeckScreen, PlayMenu, Help, Interface and EntityUpdates classes. All of these classes except Interface class have several ActionElement objects to recognize the actions of the user. These actions will be sent to the Interface class, and Interface class notifies the Game Controller subsystem. EntityUpdates class takes the GameEntities notifications and sends them to their corresponding classes in User Interface subsystem. Interface and EntityUpdates class are created to reduce the coupling.



Game Controller subsystem contains; GameController, BoardController, PlayerController, HeroController, InterfaceController, and EntityController classes. Interface class of the User Interface subsystem sends notifications to InterfaceController class. InterfaceController class sends the corresponding notifications to proper classes in Game Controller. GameController, BoardController, HeroController and PlayerController classes gets the data from the EntityController class, which gets the data from the Game Entities subsystem, and sends the result of the data changes to Game Entities. Then, Game Entities classes will notify the User Interface classes to change their user interfaces accordingly. In this process, to reduce the coupling, we need the EntityController class, which is the only class that connects with the GameEntitites subsystem to reduce coupling.

In the Game Controller subsystem, GameController class only controls the main game elements, such as the turn of the game, actions of the minions and so on. Board class only controls the board of the game, and Player class controls the actions of the players. Therefore, if a minion attacks to a Player,

GameController will notify the PlayerController. To reduce the complexity, all of the Game Controller classes connected to each other, which will increase the coherence.
Game Entities subsystem contains all of the model classes; Card, Deck, Hero, Deck, and ControllerConnector and InterfaceConnector classes. The ControllerConnector class is used



to get information from the model classes and sends them to the Game Controller subsystem as needed. The InterfaceConnector class is used to update the User Interface subsystem classes when the model classes are changed. Both ControllerConnector and InterfaceConnector classes are used in order to reduce coupling of the system

6.3 Architectural Styles

In our project we will be using both MVC (Model View Controller) style and also layered architecture. The reason we are using MVC is it really fits our design because we divided our system into 3 subsystems and because of this; MVC is really good for our design purposes.

Our subsystems are called User Interface subsystem, Game Controller subsystem and Game Entities subsystem. In our design, the User Interface subsystem will be the "view" in the MVC architectural style and it will handle all the interface related operations and will trigger the Game Controller subsystem. Game Controller will be containing all the game mechanics and all game related services. Game Controller subsystem will be our "control" part in the MVC style. Game controller subsystem will change the Game Entities subsystem accordingly and Game Entities will be our "model" and it will update the User Interface subsystem classes accordingly. Therefore, we have a triangular design MVC which suits our project perfectly.

Also, during the implementation of our project, MVC gives us the option to work on different parts simultaneously by different people without having much difficulty and also because we separated the system into 3 smaller subsystems, our project will fit the object oriented design.

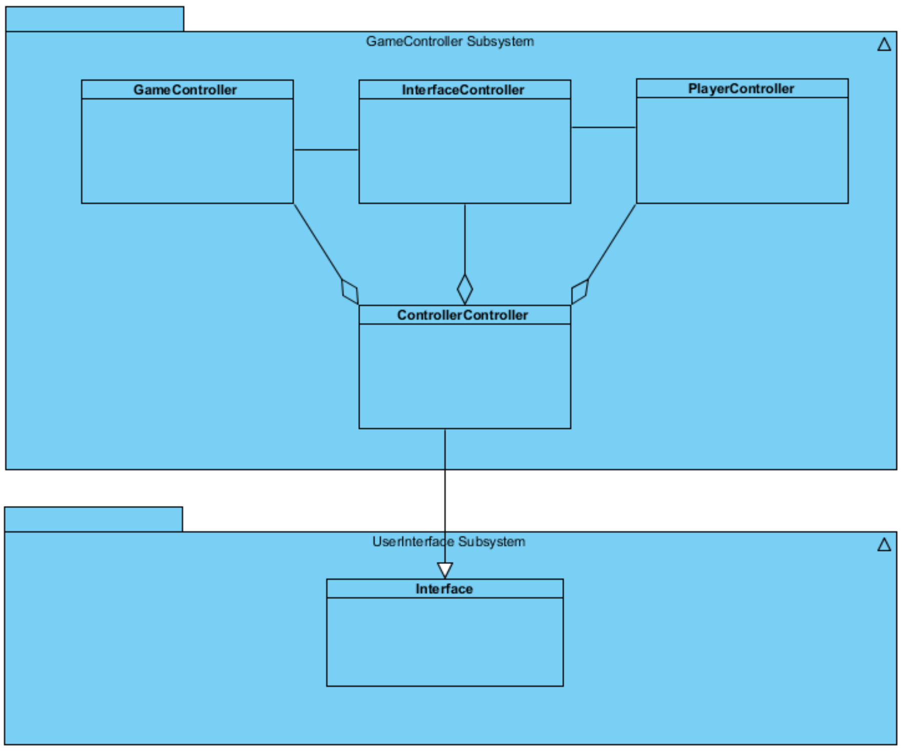
7. Object-Design

7.1 Pattern Applications

7.1.1 Façade Pattern

Façade pattern is a structural pattern which is used to simplify a complex body of code. Façade provides a simplified and easy to use interface for a complex system and by doing so it makes software systems easier to use easier to understand and . It provides convenient methods for the system which uses the methods in the libraries and packages.

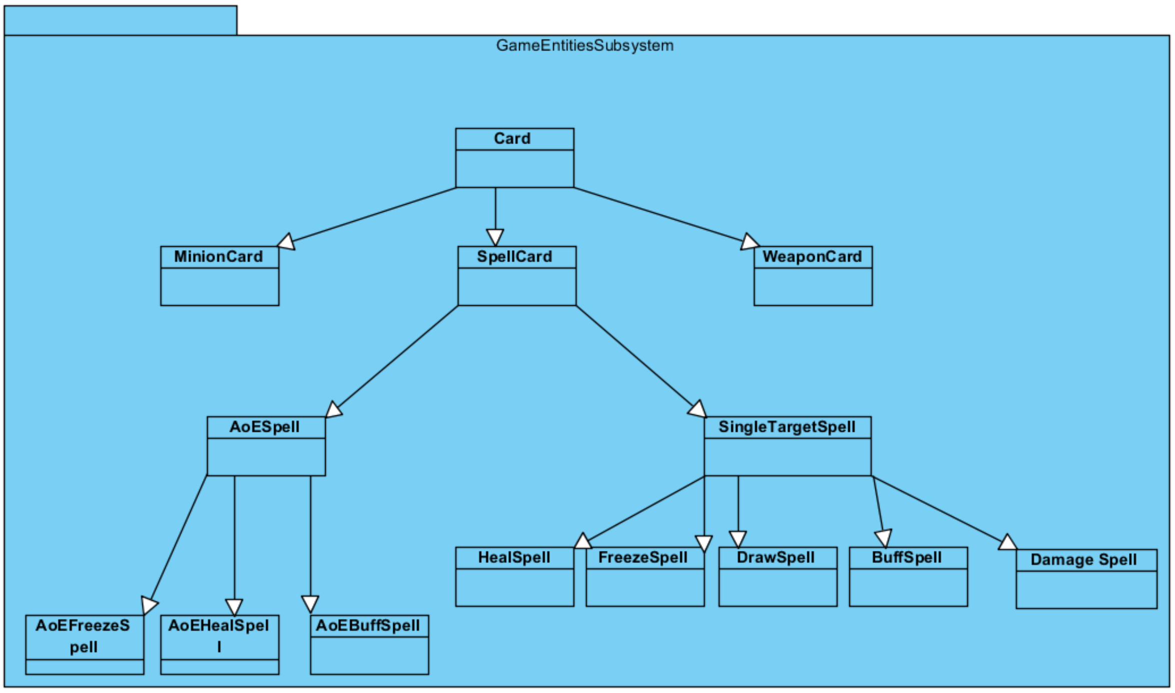
In our controllerConnector class we used façade pattern to manage all the game mechanic related classes and operations in one façade, so all of the controller classes(i.e entity controller, interface conttroller, game controller) will be controlled by this controllerConnector.



7.1.2 **Template Pattern**(Card Inheritance tree)

Template method pattern is a behavioral design pattern. This design pattern requires a class with abstract and non-abstract and several classes that inherit these abstract methods. For example a parent class provides a template for its children, but the parent class determines the algorithm. So every program uses polymorphism is eventually using thetemplate method pattern.

We used the template pattern to manage the inheritance relations in cards. The template pattern was perfect for our design because we had to manage alot of sub-classes because of the inheritance relationships. The main "card" class is a template for all of our card classes so this design will provide us neat interface to manage all of our sub-classes and it will give us the opportuniy to use polymorphism to its fullest.



7.1.3 Factory Pattern

The factory pattern is a creational design pattern. Basically all this design pattern does is, instead of creating an object every time we need that class, it only creates one instance of that class. Every time the program needs an object it just clones that base "object" modifies it and uses that clone for the program.

In our design we will be creating the "card" objects by factory pattern. This will provide us the opportunity to easily "clone" the cards so it will be easier for us to create and use our card objects.

7.1.4 Singleton Pattern(Entity Controller)

Singleton pattern is a creational pattern. It restricts the creation of an objecto to 1. This style is useful when we only need 1 object to access the data in that class.

We used singleton pattern only in Entity Controller class, because we just need one object of the EntityController class to controll the needed data and methods and it would beeasier to reach the data we need through one object of EntityController class.

7.2 Class Interfaces

In this section, every class is provided below.

Player class represents the players of the game.

Player
-opponent : Player -hero : Hero -hisTurn : boolean -hand : Card[] -playedCards : Card[] -selectedMinion : MinionCard -selectedHero : Hero -targetMinion : MinionCard -targetHero : Hero

+Player(hero:Hero)
+getOpponent() : Player
+setOpponent(opponent:Player)
+setHisTurn(state:boolean)
+getHisTurn() : boolean
+getPlayedCards() : Card[]
+setHero(hero:Hero)
+getHero() : Hero
+setSelectedMinion(selectedMinion:MinionCard)
+getSelectedMinion() : MinionCard
+setSelectedHero(selectedHero:Hero)
+getSelectedHero() : Hero
+getTargetMinion() : Card
+setTargetMinion(targetCard:Card)
+getTargetHero() : Hero
+setTargetHero(targetHero:Hero)
+attack(Card)
+select(Card)
+attackHero()
+drawCard()
+createPlayer()
+playCard()
+selectDeck()

Hero class represents the heroes and their in-game abilities.

Hero
-name : String -health : int -armor : int -mana : int -attack : int

+Hero(name:String)
+Hero(name:String, health:int)
+setName(name:String)
+getName() : String
+getHealth() : int
+setAttack(number:int)
+getAttack() : int
+attacked(damage:int)
+getArmor() : int
+getMana() : int
+useMana(number:Mana)
+increaseMana()
+increaseMana(number:int)
+increaseArmor()
+increaseArmor(number:int)
+useAbility()

Board is the class that represents the board of the game where the minions are played.

Board
-cards : MinionCard[]

+Board()
+getCards()
+playCard(card:Card)

Deck is the class that represents the card decks of the heroes. A deck consists of 30 Cards.

Deck
-cards : Card[]

+Deck()
+drawCard() : Card
+addCard(card:Card)

Card is an abstract class that holds the general methods and variables of the cards. It is the super class of MinionCard, SpellCard, WeaponCard.

Card
-name : String -text : String -mana : int -classType : String -image : BufferedImage
+setName() : String +setText() : String +setMana() : int +setClassType() : String +getName() : String +getText() : String +getMana() : int +getClassType() : String +setImage(image:BufferedImage) +getImage() : BufferedImage

WeaponCard is the class that represents the weapon cards of the game.

WeaponCard
-durability : int -damage : int
+WeaponCard(durability:int, damage:int) +getDamage() : int +getDurability() : int

SpellCard is an abstract class that represents the spell cards of the game. It is the superclass for AoESpell, SingleTargetSpell and NonTargetSpell.

SpellCard
-isPlayable : boolean -player : Player

+setIsPlayable(state:boolean)
+getIsPlayable() : boolean
+setPlayer(player:Player)
+playEffect()

SingleTargetSpell is an abstract class that represents the spell cards that targets single card or character. It is the superclass of HealSpell, FreezeSpell, DrawSpell, BuffSpell, DamageSpell.

SingleTargetSpell
-number1 : int -number2 : int

+setNumber1(number1:int)
+getNumber1() : int
+setNumber2(number2:int)
+getNumber2() : int

HealSpell is a class that represents the spell cards that heals a single target.

HealSpell
+playEffect()

FreezeSpell is a class that represents the spell cards that freezes a single target.

FreezeSpell
+playEffect()

DrawSpell is a class that represents the spell cards that makes card draw and buffs or damages a single minion.

DrawSpell
+playEffect()

BuffSpell is a class that represents the spell cards that buffs a single minion.

BuffSpell
+playEffect()

DamageSpell is a class that represents the spell cards that damages a single target.

DamageSpell
+playEffect()

AoESpell is an abstract class that represents the spell cards that effects several minions. It is the superclass of AoEHealSpell, AoEFreezeSpell, AoEBuffSpell, AoEDamageSpell.

AoESpell
-number1 : int -number2 : int

+setNumber1(number1:int)
+getNumber1() : int
+getNumber2() : int
+setNumber2(number2:int)

AoEHealSpell is a class that represents the spell cards that heals minions on the board.

AoEHealSpell
+playEffect()

AoEFreezeSpell is a class that represents the spell cards that freezes minions on the board.

AoEFreezeSpell
+playEffect()

AoEBuffSpell is a class that represents the spell cards that buffs minions on the board.

AoEBuffSpell
+playEffect()

AoEDamageSpell is a class that represents the spell cards that damages minions on the board.

AoEDamageSpell
+playEffect()

MinionCard is a class that represents the minions of the game.

MinionCard
-damage : int -health : int

+setDamage(damage:int)
+getDamage() : int
+setHealth(health:int)
+getHealth() : int

ControllerConnector is a class that gets information from the model classes and sends them to the Game Controller subsystem as needed.

ControllerConnector
- player1 : Player - player2 : Player - board : Board
+ ControllerConnector(player1:Player, player2:Player, board:Board) +getPlayer1() : Player + getBoard() : Board + getPlayer2() : Player +

EntityController is a class that connects with the Game Entities subsystem.

EntityController
- controllerConnector : ControllerConnector - entityController : EntityController
- EntityController(controllerConnector:ControllerCo nnector) + getEntityController() : EntityController +getControllerConnector():ControllerConnector

InterfaceController is a class that sends the corresponding notifications to proper classes in Game Controller subsystem.

InterfaceController
- interfaceConnector : Interface -interfaceController : interfaceController
- InterfaceController(interfaceConnector:Interface) +getInterfaceConnector() : InterfaceConnector +getInterfaceController() : InterfaceController

GameController is a class that controls the deck creation and game interface.

GameController
- entityController : EntityController - interfaceController : InterfaceController - playerController : PlayerController
+ GameController() + startGame() + setPlayerController(playerController:PlayerContro ller) + getPlayerController() : PlayerController

PlayerController is a class that controls the in-game player features.

PlayerController
- entityController : EntityController - interfaceController : InterfaceController - heroController : HeroController - turnTime : int - turn : int

+ getHeroController()
+ PlayerController()
+ updateCards()
+ resetTurnTime()
+ setTurn(turn:int)
+ newGame()
+ selectDeck(player:Player)
+playCard()
+ concede(player:Player)
+ isGameEnd(player:Player)

HeroController is a class that controls the in-game hero features.

HeroController
- entityController : EntityController - interfaceController : InterfaceController - boardController : BoardController - mana : int - health : int

+ setBoardController()
+HeroController()
+ increaseMana()
+ increaseMana(int amount)
+ isGameEnd()
+ useAbility()

BoardController is a class that controlls in-game board activities.

BoardController
- entityController : EntityController - interfaceController : InterfaceController - playedMinions : Card[]

+ playMinion(player:Player, minion:MinionCard)
+ BoardController()
+setEntityController()
+ setInterfaceController

7.3 Specifying Contracts using OCL

Player:

1. **context** Player:: select(Card)

post:

self.selectedCard+1 = Card.selectedCard@pre

//After each selecting card number of selected cards will be incremented.

2. **context** Player:: playCard()

post:

self.playedCard[size+1] = playedCard[size]@pre

//After calling card size of playedCard array will be incremented contrast to previous one.

3. **context** Player::getPlayedCards() : Card[]

post :

result = playedCards[]

//After each calling of getPlayedCards method, it should be concluded with cards played.

4. **context** Player::getHero() : Hero

post :

result = hero

//After each calling of getHero method, it should be concluded with hero.

5. **context** Player::setHero(hero = Hero)

post :

self.Hero = hero

// After each calling of setHero method, it should be concluded with hero.

6. **context** Player::getHisTurn() : boolean

post :

result = hisTurn

//After each calling of getHisTurn method, it should be concluded with histurn.

7. **context** Player::setHisTurn(state:boolean)

post :

self.hisTurn = state

//After each calling of setHisTurn method, it should be concluded with state of histurn.

8. **context** Player::getOpponent() : Player

pre:

result = player.opponent

//All calling of getOpponent method, when player gets oppponent.

9. **context** Player::setOpponent(opponent:Player)

pre:

self.Opponent = state

//All calling of setHisTurn method, when state is selected as state of opponent.

10. **context** Player:: drawCards() : boolean

pre:

(player.selectCard() > self.selectCard())

post:

player.deck.add(self.deck.getCard(0)) && self.deck.remove(0)

// when player card selection is avaible, draw card is usable after cards is selected and put the board

<u>**Hero:**</u>

11. **context** Hero::getHeroName() : String

post :

result = heroName

// After each calling of getHeroName method, it should be concluded with hero name.

12. **context** Hero::setHeroName(name = String)

post :

self.userName = name

// After each calling of setHeroName method, it should be concluded with name.

13. **context** Hero::getHealth() : int

post :

result = health

// After each calling of getHealth method, it should be concluded with health.

14. **context** Hero::getArmor() : int

post :

result = armor

// After each calling of getArmor method, it should be concluded with armor.

15. **context** Hero::getMana() : int

post :

        result = mana

// After each calling of getMana method, it should be concluded with hero mana.

16. **context** Hero::getAttack () : int

post :

        result = attack

// After each calling of getAttack method, it should be concluded with hero attack.

17. **context** Hero::setAttack(number = int)

post :

        self.attack = number

// After each calling of setAttack method, it should be concluded with number.

18. **context** Hero::useAbility()

pre :

        mana-2 = mana@pre

// when programme use ability it takes point from mana same level 2

19. **context** Hero::increaseArmor()

post :

        armor+1 = armor@pre

// After calling of increaseArmor method, armor is incremented by 1.

20. **context** Hero::increaseArmor(number: int)

post :

        armor+number = armor@pre

// After calling of increaseArmor method, armor is incremented by number.

21. **context** Hero::increaseMana()

post :

mana+1 = mana@pre

// After calling of increaseMana method, mana is incremented by 1.

22. **context** Hero::increaseMana(number: int)

post :

mana+number = mana@pre

// After calling of increaseMana method, mana is incremented by number.

23. **context** Hero::useMana (number:Mana)

pre:

Player.playCard()&& Hero.mana--;

// when play card and mana is decremented, mana is used.

24. **context** Hero::attacked(damage: int)

post :

hero.getHealth() - damage = hero.getHealth() @pre

// After calling of attacked health is decremented by damage size.

Deck:

25. **context** Deck::addCard(card: Card)

post :

self.add(self.size-1,card)

// After calling of add card it adds with size dicremented by 1 to cards

Card:

26. **context** Card::setCard():void

post:

return =set.ImageFile().setHero().JPanel actionPerformed()

// After calling of set card it adds card as image file

27. **context** Card::setText():String

post:

self.text

SpellCard:

28. **context** SpellCard::getIsPlayable():boolean

post:

    result = isPlayable

// After calling of getIsPlayable it gets isPlayable variable

29. **context** SpellCard::setIsPlayable(state: boolean)

post:

        self.isPlayable = state

// After calling of setIsPlayable it sets isPlayable variable as state

30. **context** SpellCard::setPlayer(player: Player)

post:

        self.Player = player

// After calling of setPlayable it sets player

8. Conclusion and Lessons Learned

Our project is called Hearthstone: Heroes of Warcraft it is basically a card game where players use various cards to defeat their opponents.

In the first step of our project we made a “Requirements report” in which we clarified basically our requirements for our project which are the functional and non-functional requirements, constraints, scenarios, use case models and user interfaces. Also we thought and worked in our code to be precise while writing those requirements.

In the next step we we handled the design of the project, and in this step we started to see our work paying off. In this part basically we made the skeleton of our project and talked about the functionality of our game, how the classes and sub-systems interact and which architectural styles we are gonna use in our game.

Last but not least we finished our report by explaining all the classes sub-systems and how classes interact in our final report. In this report we really made sure how our code is gonna work and thought really hard on how to implement our game.

In this project we tried to make an already existing game because we thought that Hearthstone was the perfect game to test our skills in object-oriented software programming. The game contained high class interactions and it was perfect project to apply the patterns we learned through out the course. Also the game was a fun game so we had fun while we wrote the reports and also thought about the code. Also as a group we were curious about how games are created and what type of problems the real engineers face while they were working on a game project. We also learned the UML language and how to use it to express our ideas while working on our project. But our main aim was to make our own game and to find out maybe we can also make our game and sell it in the market in the future.

It was overall a good chalenge for us and we think that this project took us a step closer to become a real software engineers.