Ankara Yıldırım Beyazıt University
Department of Computer Engineering

CENG 201 – Object Oriented Programming
Course Project

G10: Mini Yu-Gi-Oh! Game

Class Design

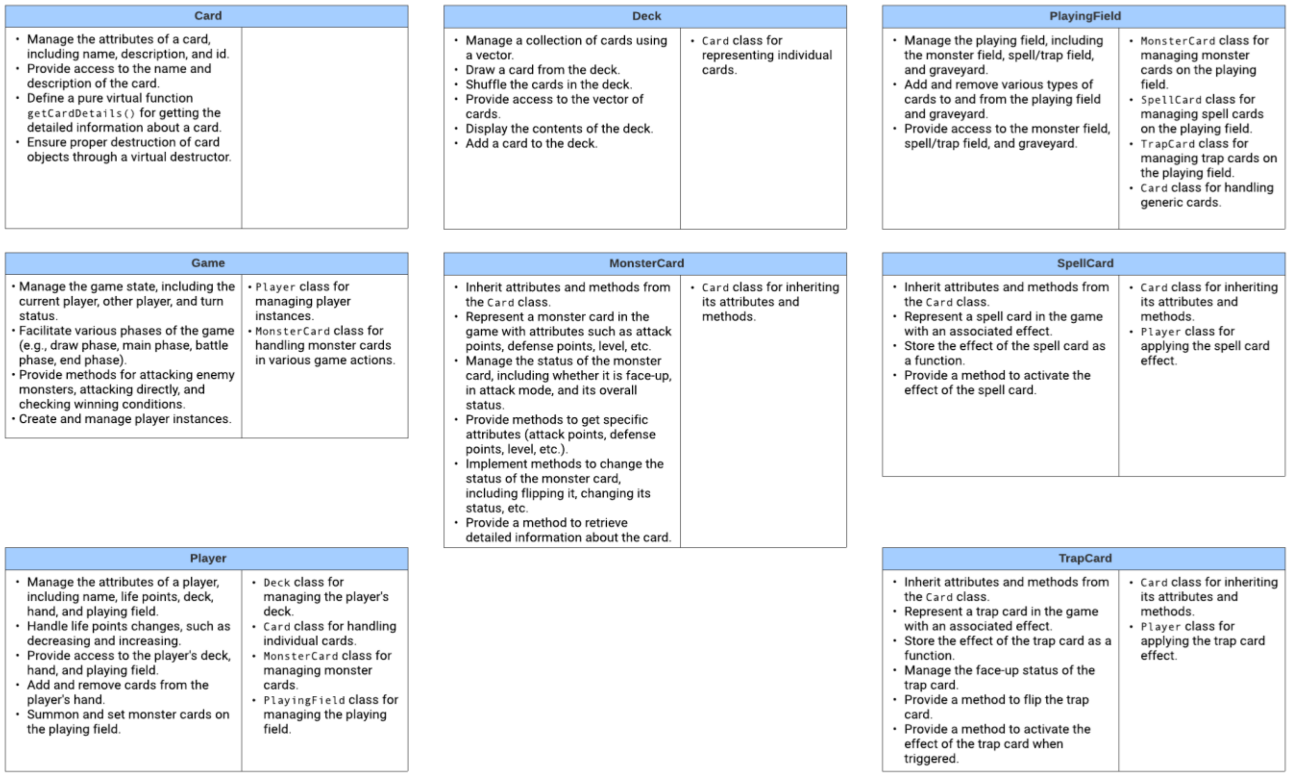
Emre Demirbaş, 21050111069
Mehmet Ali Yılmaz, 21050111057
Ömer Faruk Özüyağlı, 21050111047
Muhammed Esat Çelebi, 21050111034

Instructor: Muhammed Abdullah Bülbül
Teaching Assistant: Elif Şanlıalp
Date: 12/12/2023

1. Introduction

This report summarizes the design and implementation of the Mini Yu-Gi-Oh! Game. This project involves creating various C++ classes that model important components of the game such as cards, players, decks, and playing fields. The goal is express the game's structure and functionality clearly and concisely through object-oriented design principles. This report begins by presenting Class Responsibility Collaboration (CRC) cards for each class, outlining their roles, responsibilities, and collaboration. Next, you will see a class diagram that shows the relationships and interactions between classes.

2. Class-Responsibility-Collaboration (CRC) Cards



Card	
• Manage the attributes of a card, including name, description, and id. • Provide access to the name and description of the card. • Define a pure virtual function `getCardDetails()` for getting the detailed information about a card. • Ensure proper destruction of card objects through a virtual destructor.	

Deck	
• Manage a collection of cards using a vector. • Draw a card from the deck. • Shuffle the cards in the deck. • Provide access to the vector of cards. • Display the contents of the deck. • Add a card to the deck.	• `Card` class for representing individual cards.

PlayingField	
• Manage the playing field, including the monster field, spell/trap field, and graveyard. • Add and remove various types of cards to and from the playing field and graveyard. • Provide access to the monster field, spell/trap field, and graveyard.	• `MonsterCard` class for managing monster cards on the playing field. • `SpellCard` class for managing spell cards on the playing field. • `TrapCard` class for managing trap cards on the playing field. • `Card` class for handling generic cards.

Game	
• Manage the game state, including the current player, other player, and turn status. • Facilitate various phases of the game (e.g., draw phase, main phase, battle phase, end phase). • Provide methods for attacking enemy monsters, attacking directly, and checking winning conditions. • Create and manage player instances.	• `Player` class for managing player instances. • `MonsterCard` class for handling monster cards in various game actions.

MonsterCard	
• Inherit attributes and methods from the `Card` class. • Represent a monster card in the game with attributes such as attack points, defense points, level, etc. • Manage the status of the monster card, including whether it is face-up, in attack mode, and its overall status. • Provide methods to get specific attributes (attack points, defense points, level, etc.). • Implement methods to change the status of the monster card, including flipping it, changing its status, etc. • Provide a method to retrieve detailed information about the card.	• `Card` class for inheriting its attributes and methods.

SpellCard	
• Inherit attributes and methods from the `Card` class. • Represent a spell card in the game with an associated effect. • Store the effect of the spell card as a function. • Provide a method to activate the effect of the spell card.	• `Card` class for inheriting its attributes and methods. • `Player` class for applying the spell card effect.

Player	
• Manage the attributes of a player, including name, life points, deck, hand, and playing field. • Handle life points changes, such as decreasing and increasing. • Provide access to the player's deck, hand, and playing field. • Add and remove cards from the player's hand. • Summon and set monster cards on the playing field.	• `Deck` class for managing the player's deck. • `Card` class for handling individual cards. • `MonsterCard` class for managing monster cards. • `PlayingField` class for managing the playing field.

TrapCard	
• Inherit attributes and methods from the `Card` class. • Represent a trap card in the game with an associated effect. • Store the effect of the trap card as a function. • Manage the face-up status of the trap card. • Provide a method to flip the trap card. • Provide a method to activate the effect of the trap card when triggered.	• `Card` class for inheriting its attributes and methods. • `Player` class for applying the trap card effect.

Figure 1. CRC Cards

3. Class Diagram

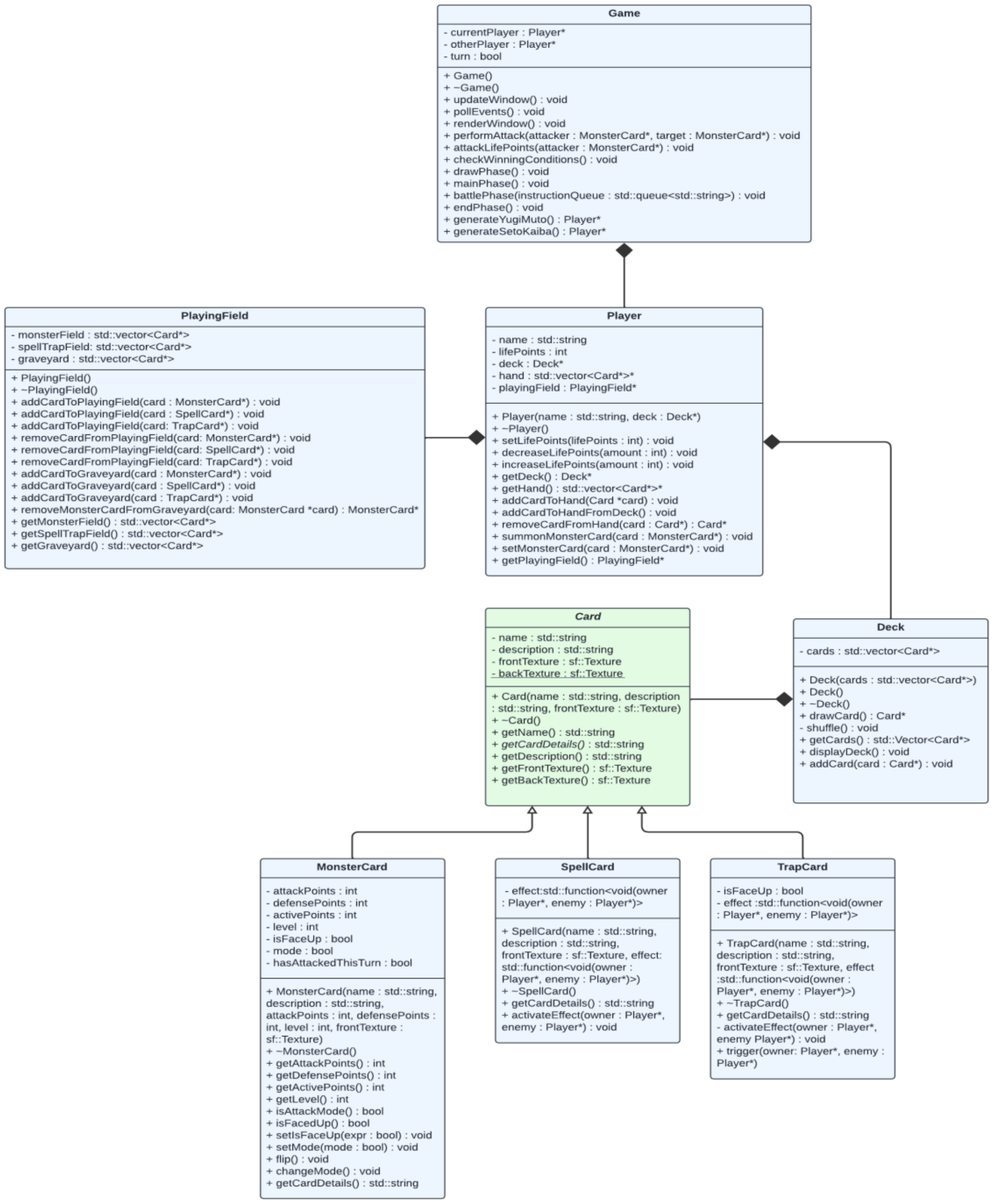

Game
- currentPlayer : Player*
- otherPlayer : Player*
- turn : bool
+ Game()
+ ~Game()
+ updateWindow() : void
+ pollEvents() : void
+ renderWindow() : void
+ performAttack(attacker : MonsterCard*, target : MonsterCard*) : void
+ attackLifePoints(attacker : MonsterCard*) : void
+ checkWinningConditions() : void
+ drawPhase() : void
+ mainPhase() : void
+ battlePhase(instructionQueue : std::queue<std::string>) : void
+ endPhase() : void
+ generateYugiMuto() : Player*
+ generateSetoKaiba() : Player*
PlayingField
- monsterField : std::vector<Card*>
- spellTrapField: std::vector<Card*>
- graveyard : std::vector<Card*>
+ PlayingField()
+ ~PlayingField()
+ addCardToPlayingField(card : MonsterCard*) : void
+ addCardToPlayingField(card : SpellCard*) : void
+ addCardToPlayingField(card: TrapCard*) : void
+ removeCardFromPlayingField(card: MonsterCard*) : void
+ removeCardFromPlayingField(card: SpellCard*) : void
+ removeCardFromPlayingField(card: TrapCard*) : void
+ addCardToGraveyard(card : MonsterCard*) : void
+ addCardToGraveyard(card : SpellCard*) : void
+ addCardToGraveyard(card : TrapCard*) : void
+ removeMonsterCardFromGraveyard(card: MonsterCard *card) : MonsterCard*
+ getMonsterField() : std::vector<Card*>
+ getSpellTrapField() : std::vector<Card*>
+ getGraveyard() : std::vector<Card*>
Player
- name : std::string
- lifePoints : int
- deck : Deck*
- hand : std::vector<Card*>*
- playingField : PlayingField*
+ Player(name : std::string, deck : Deck*)
+ ~Player()
+ setLifePoints(lifePoints : int) : void
+ decreaseLifePoints(amount : int) : void
+ increaseLifePoints(amount : int) : void
+ getDeck() : Deck*
+ getHand() : std::vector<Card*>*
+ addCardToHand(Card *card) : void
+ addCardToHandFromDeck() : void
+ removeCardFromHand(card : Card*) : Card*
+ summonMonsterCard(card : MonsterCard*) : void
+ setMonsterCard(card : MonsterCard*) : void
+ getPlayingField() : PlayingField*
Card
- name : std::string
- description : std::string
- frontTexture : sf::Texture
- backTexture : sf::Texture
+ Card(name : std::string, description : std::string, frontTexture : sf::Texture)
+ ~Card()
+ getName() : std::string
+ getCardDetails() : std::string
+ getDescription() : std::string
+ getFrontTexture() : sf::Texture
+ getBackTexture() : sf::Texture
Deck
- cards : std::vector<Card*>
+ Deck(cards : std::vector<Card*>)
+ Deck()
+ ~Deck()
+ drawCard() : Card*
- shuffle() : void
+ getCards() : std::Vector<Card*>
+ displayDeck() : void
+ addCard(card : Card*) : void
MonsterCard
- attackPoints : int
- defensePoints : int
- activePoints : int
- level : int
- isFaceUp : bool
- mode : bool
- hasAttackedThisTurn : bool
+ MonsterCard(name : std::string, description : std::string, attackPoints : int, defensePoints : int, level : int, frontTexture : sf::Texture)
+ ~MonsterCard()
+ getAttackPoints() : int
+ getDefensePoints() : int
+ getActivePoints() : int
+ getLevel() : int
+ isAttackMode() : bool
+ isFacedUp() : bool
+ setIsFaceUp(expr : bool) : void
+ setMode(mode : bool) : void
+ flip() : void
+ changeMode() : void
+ getCardDetails() : std::string
SpellCard
- effect:std::function<void(owner : Player*, enemy : Player*)>
+ SpellCard(name : std::string, description : std::string, frontTexture : sf::Texture, effect: std::function<void(owner : Player*, enemy : Player*)>)
+ ~SpellCard()
+ getCardDetails() : std::string
+ activateEffect(owner : Player*, enemy : Player*) : void
TrapCard
- isFaceUp : bool
- effect :std::function<void(owner : Player*, enemy : Player*)>
+ TrapCard(name : std::string, description : std::string, frontTexture : sf::Texture, effect :std::function<void(owner : Player*, enemy : Player*)>)
+ ~TrapCard()
+ getCardDetails() : std::string
- activateEffect(owner : Player*, enemy Player*) : void
+ trigger(owner: Player*, enemy : Player*)

Figure 2. Class Diagram

4. Conclusion

In summary, the class design for the Mini Yu-Gi-Oh! game project follows a systematic and thoughtful approach. CRC cards effectively capture each class's responsibilities and collaboration, providing a detailed understanding of the classes' intended role in the game. Class diagrams visually represent the relationships between these classes and ensure a comprehensive overview of the system. The class diagram illustrates the relationships and interactions between classes, offering a visual representation of the game's architecture. All team members collaborated as partners while working on the report.