Informe del Proyecto de Battle Cards

Integrantes:

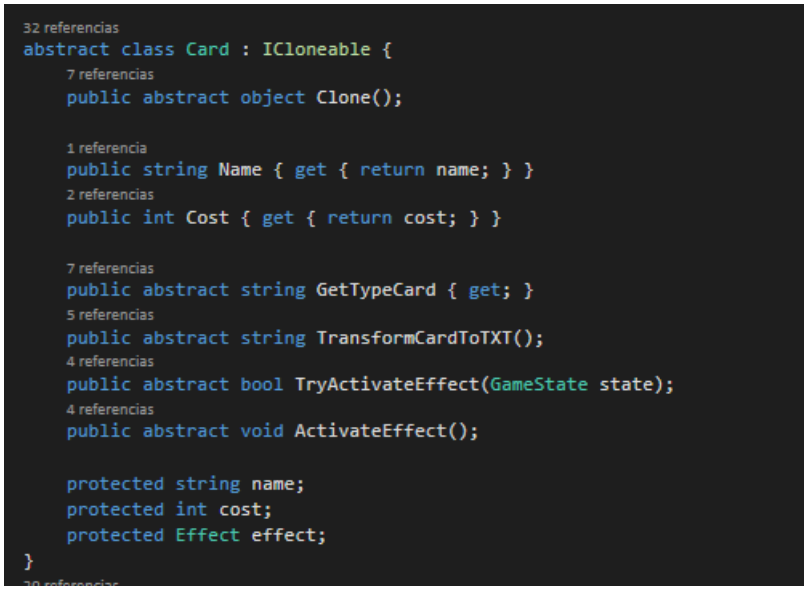
- *Raimel Daniel Romaguera Puig ……. (C113)*
- *Javier David Coroas Cintra ……. (C113)*

```
11 referencias
class Game {
    1 referencia
    public Game(Deck deck1, Deck deck2, Rules rules) {
        this.rules = rules;
        state = new GameState();
        p1 = new Persona(deck1, rules.GetMaxLife, rules.GetMaxHand, rules.GetMaxBoard);
        p2 = new Persona(deck2, rules.GetMaxLife, rules.GetMaxHand, rules.GetMaxBoard);
    }
    2 referencias
    public Game(Deck deck1, Deck deck2, Rules rules, IInteligence inteligence1) {
        this.rules = rules;
        state = new GameState();
        p1 = new Persona(deck1, rules.GetMaxLife, rules.GetMaxHand, rules.GetMaxBoard);
        p2 = new Virtual(deck2, rules.GetMaxLife, rules.GetMaxHand, rules.GetMaxBoard, inteligence1);
    }
    1 referencia
    public Game(Deck deck1, Deck deck2, Rules rules, IInteligence inteligence1, IInteligence inteligence2) {
        this.rules = rules;
        state = new GameState();
        p1 = new Virtual(deck1, rules.GetMaxLife, rules.GetMaxHand, rules.GetMaxBoard, inteligence1);
        p2 = new Virtual(deck2, rules.GetMaxLife, rules.GetMaxHand, rules.GetMaxBoard, inteligence2);
    }

    1 referencia
    public void NextTurn()...

    9 referencias
    public GameState GetState { get { return state; } }
    6 referencias
    public Player GetPlayer { get { return p; } }
    3 referencias
    public string GetNamePlayer { get { return namePlayer; } }
    1 referencia
    public string GetOpponentPlayer { get { return nameOpponent; } }
    11 referencias
    public Board GetBoardOpponent { get { return boardOpponent; } }

    public int lifeOpponent;
    public bool[] maskEffect, maskAttack;
```

La clase **Game** tiene 3 constructores los cuales indican: persona contra persona, persona contra jugador virtual y jugador virtual contra jugador virtual. El jugador virtual tiene que cumplir con la interfaz **IInteligence** que tiene que tener los métodos claves de una partida, siendo estos: invocar carta, activar efectos y atacar a la vida del oponente, a una estructura u otra carta (soldado)
El método fundamental que define la clase es *NextTurn* que no es necesario parámetros ni devuelve nada. Este hace que las variables que indican el jugador que le toque jugar se actualice y por cada turno cambie de jugador cada vez que se invoque el método, además esto también modifica el número de turno, inicializa de nuevo la vida del jugador y del oponente, como del mana, siendo esta la base que permite a una carta ser invocada.
Los arreglos booleanos maskEffect y maskAttack toman un papel importante en cada partida, donde cuando se llama al método de *NextTurn* este revisa el campo del jugador que le toque jugar y si una carta ya ha estado en el campo al menos un turno, esta puede atacar y activar su efecto (siempre que cumpla con la condición la carta va a poder activar su efecto)

Los “efectos” en este juego funcionan de la siguiente manera:
- Para poder activar un efecto tiene que cumplirse claramente su condición de activación
- Cada efecto consiste en modificar sus atributos que tienen en el campo, siendo estos los básicos como son: ataque y defensa.

A la hora de atacar una carta a otra, nos basamos en el famoso juego Shadowverse, el cual si dos cartas de tipo soldado atacan (estas tienen ataque y defensa), la defensa de una carta se le restara el ataque de la carta atacante y viceversa. Como idea “original” decidimos que para poder atacar al líder primero no puede haber en el campo una carta de tipo estructura, la cual solo tendrá defensa (puede tener efecto). Además, solo puede haber una estructura en cada campo.

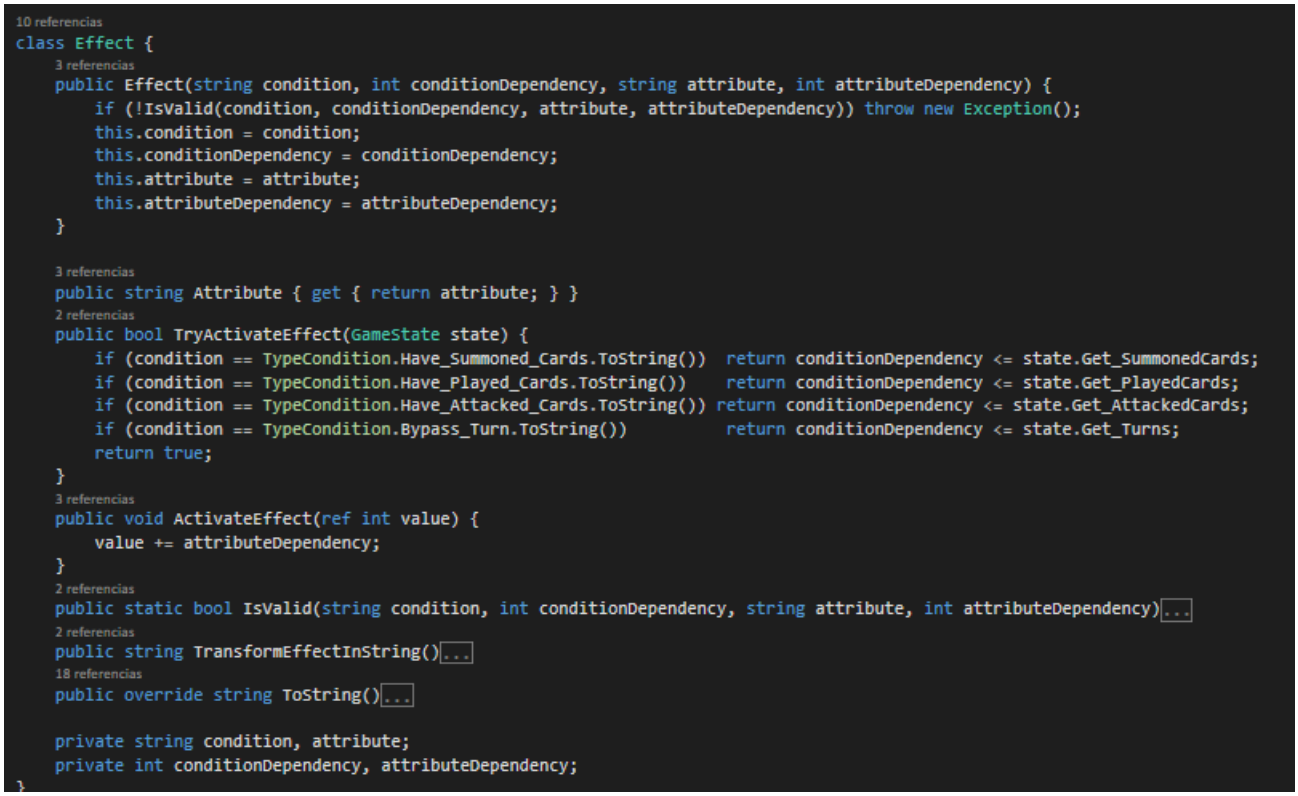


```
32 referencias
abstract class Card : ICloneable {
    7 referencias
    public abstract object Clone();

    1 referencia
    public string Name { get { return name; } }
    2 referencias
    public int Cost { get { return cost; } }

    7 referencias
    public abstract string GetTypeCard { get; }
    5 referencias
    public abstract string TransformCardToTXT();
    4 referencias
    public abstract bool TryActivateEffect(GameState state);
    4 referencias
    public abstract void ActivateEffect();

    protected string name;
    protected int cost;
    protected Effect effect;
}
```

La clase abstracta **Card**, como bien su nombre indica, es el molde de toda carta que se quiera crear en el juego, son los atributos obligatorios que tiene que tener. Sus métodos más importantes son *TransformCardToTXT*, *ActivateEffect* y *TryActivateEffect*
Lo cual para poder activar cualquier efecto se ejecuta *ActivateEffect* y este tiene dentro *TryActivateEffect* lo cual revisa la condición según un estado de juego. Ambos métodos están públicos para que, en la visual, si se desea puede comprobarse si se puede ejecutar el efecto, es decir si cumple la condición, y además poder activar el efecto de la carta
El método *TransformCardToTXT* es el encargado de cuando se cree una carta este método guarda los datos de la carta en un archivo .txt

Estructura del .txt de la carta tipo Soldier: *name + cost + attack + defense + effect*
Estructura del .txt de la carta tipo Struct: *name + cost + defense + effect*
Estructura del .txt de la clase tipo Effect: *condition + value + attribute + value*

Cuando se crea un efecto a una carta de tipo Struct (estructura) puede incluírsele que se le modifique el atributo de ataque, pero como esta no tiene es como si no tuviera efecto la carta.



```
10 referencias
class Effect {
    3 referencias
    public Effect(string condition, int conditionDependency, string attribute, int attributeDependency) {
        if (!IsValid(condition, conditionDependency, attribute, attributeDependency)) throw new Exception();
        this.condition = condition;
        this.conditionDependency = conditionDependency;
        this.attribute = attribute;
        this.attributeDependency = attributeDependency;
    }

    3 referencias
    public string Attribute { get { return attribute; } }
    2 referencias
    public bool TryActivateEffect(GameState state) {
        if (condition == TypeCondition.Have_Summoned_Cards.ToString())  return conditionDependency <= state.Get_SummonedCards;
        if (condition == TypeCondition.Have_Played_Cards.ToString())    return conditionDependency <= state.Get_PlayedCards;
        if (condition == TypeCondition.Have_Attacked_Cards.ToString()) return conditionDependency <= state.Get_AttackedCards;
        if (condition == TypeCondition.Bypass_Turn.ToString())          return conditionDependency <= state.Get_Turns;
        return true;
    }
    3 referencias
    public void ActivateEffect(ref int value) {
        value += attributeDependency;
    }
    2 referencias
    public static bool IsValid(string condition, int conditionDependency, string attribute, int attributeDependency)...
    2 referencias
    public string TransformEffectInString()...
    18 referencias
    public override string ToString()...

    private string condition, attribute;
    private int conditionDependency, attributeDependency;
}
```

La clase **Effect**, es independiente de la clase **Card**, ya que por sí sola esta clase tiene su propia y única responsabilidad, que es activar el efecto siempre y cuando se cumpla la condición. Cada efecto tiene un valor que depende de la condición (conditionDependency) y un valor que modifica el atributo (attributeDependency) esto significa que para cumplir la condición tiene que tener como mínimo el valor de la condición y en caso del valor del atributo, lo modifica en ese valor.
El método estático *IsValid* es sumamente importante, ya que verifica que un efecto puede considerarse valido siguiendo algunas características del juego. Ejemplo: verifica que el string attribute indique un atributo de una carta “Attack” o “Defense”

```
10 referencias
class Action {
    3 referencias
    public static void Summon(Board board, Hand hand, int posBoard, int posHand, ref int mana) {
        board[posBoard] = hand[posHand];
        mana -= board[posBoard].Cost;
        hand[posHand] = null;
    }
    1 referencia
    public static void ActivateEffect(Board board, int posBoard, ref bool[] mask) {
        board[posBoard].ActivateEffect();
        mask[posBoard] = false;
    }
    2 referencias
    public static void Attack(Board board, Board boardOpponent, int posBoard, int posBoardOpponent, ref bool[] mask) {
        ((Soldier)boardOpponent[posBoardOpponent]).defense -= ((Soldier)board[posBoard]).attack;
        ((Soldier)board[posBoard]).defense -= ((Soldier)boardOpponent[posBoardOpponent]).attack;
        mask[posBoard] = false;
        if (((Soldier)board[posBoard]).defense <= 0) board[posBoard] = null;
        if (((Soldier)boardOpponent[posBoardOpponent]).defense <= 0) boardOpponent[posBoardOpponent] = null;

    }
    2 referencias
    public static void Attack(Board board, Board boardOpponent, int posBoard, ref bool[] mask) {
        ((Struct)boardOpponent[0]).defense -= ((Soldier)board[posBoard]).attack;
        mask[posBoard] = false;
        if (((Struct)boardOpponent[0]).defense <= 0) boardOpponent[0] = null;
    }
    2 referencias
    public static void Attack(Board board, int posBoard, ref bool[] mask, ref int life) {
        life -= ((Soldier)board[posBoard]).attack;
        mask[posBoard] = false;
    }
}
```

En la clase **Action** independientemente de si el jugador que le toque el turno, puede interactuar con alguno de estos métodos
Cada método ejecuta la lógica de la partida. El método *Summon* elimina la carta de la mano y la pone en una posición del campo y luego se resta el coste de la carta al mana del jugador. El método *ActivateEffect* intenta activar el efecto de la carta en una posición.
Los diferentes (3) tipos de ataque caracterizan también la partida, siendo estos atacar al líder, restando el ataque que tiene una carta a la vida del oponente, atacar a una estructura que es bajarle la defensa a la estructura según el ataque de una carta y por último el combate entre dos cartas que tienen ataque, que es bajarse la defensa hasta que baje de 0. Estos métodos de ataque tienen algo en común, en bajar la defensa o la vida del objetivo, y si la defensa o la vida baja de 0, es una destrucción de la carta o la derrota del oponente
Estos métodos, interactúan fuertemente con los métodos de la clase **TryAction** lo cual para efectuar un método de **Action** tiene que ejecutarse su mismo en **TryAction** para evitar que lance excepciones no deseadas el método de **Action**

```
10 referencias
class TryAction {
    3 referencias
    public static bool Summon(Board board, Hand hand, int posBoard, int posHand, int mana) {
        if (!IsValid(board.Length, posBoard)) return false;
        if (!IsValid(hand.Length, posHand)) return false;
        if (posBoard != 0 && hand[posHand] is Struct) return false;
        if (board[posBoard] != null) return false;
        if (hand[posHand] == null) return false;
        if (hand[posHand].Cost > mana) return false;
        return true;
    }
    1 referencia
    public static bool ActivateEffect(Board board, int posBoard, bool[] mask) {
        if (!IsValid(board.Length, posBoard)) return false;
        if (board[posBoard] == null) return false;
        if (!mask[posBoard]) return false;
        return true;
    }
    2 referencias
    public static bool Attack(Board board, Board boardOpponent, int posBoard, int posBoardOpponent, bool[] mask) {
        if (!IsValid(board.Length, posBoard)) return false;
        if (!IsValid(boardOpponent.Length, posBoardOpponent)) return false;
        if (!mask[posBoard]) return false;
        return (board[posBoard] is Soldier && boardOpponent[posBoardOpponent] is Soldier);
    }
    2 referencias
    public static bool Attack(Board board, Board boardOpponent, int posBoard, bool[] mask) {
        if (!IsValid(board.Length, posBoard)) return false;
        if (!mask[posBoard]) return false;
        return (board[posBoard] is Soldier && boardOpponent[0] is Struct);
    }
    2 referencias
    public static bool Attack(Board board, int posBoard, bool[] mask) {
        if (!IsValid(board.Length, posBoard)) return false;
        if (!mask[posBoard]) return false;
        return (board[posBoard] is Soldier);
    }
    7 referencias
    private static bool IsValid(int length, int pos) => (0 <= pos && pos < length);
}
```

```
7 referencias
class Virtual : Player {
    3 referencias
    public Virtual(Deck deck, int maxLife, int maxHand, int maxBoard, IInteligence inteligence) {
        this.deck = deck;
        life = maxLife;
        mana = 0;
        hand = new Hand(maxHand);
        board = new Board(maxBoard);
        this.inteligence = inteligence;
    }

    1 referencia
    public void Summon(GameState state) {
        inteligence.Summon(board, hand, ref mana, state);
    }
    1 referencia
    public void ActivateEffect(ref bool[] mask, GameState state) {
        inteligence.ActivateEffect(board, ref mask, state);
    }
    1 referencia
    public void Attack(Board boardOpponent, ref bool[] mask, ref int life, GameState state) {
        inteligence.Attack(board, boardOpponent, ref mask, ref life, state);
    }

    private IInteligence inteligence;
}
4 referencias
```

La clase **Virtual**, que hereda claramente de **Player**, tiene un constructor que tiene como uno de los parámetros **IInteligence**. Esta traza la estrategia del jugador virtual, ya sea en su forma de invocar, en el orden que activa los efectos la carta y el tipo de ataque que desarrolla este contra el jugador oponente.
A continuación, mostraremos una de las estrategias que puede tener el jugador virtual.

```
4 referencias
class FullDefenseFullAttack : IInteligence
    2 referencias
    public void Summon(Board board, Hand hand, ref int mana, GameState state) {
        if (!board.HaveStruct() && hand.HaveStruct() && TryAction.Summon(board, hand, 0, hand.GetStructPosition(), mana)) {
            Action.Summon(board, hand, 0, hand.GetStructPosition(), ref mana);
            state.Increase_SummonedCards();
        }
        if (board.GetEmptyPosition() != -1) {
            for (int i = 0; i < hand.Length; i++) {
                if (TryAction.Summon(board, hand, board.GetEmptyPosition(), i, mana)) {
                    Action.Summon(board, hand, board.GetEmptyPosition(), i, ref mana);
                    state.Increase_SummonedCards();
                }
            }
        }
    }
    2 referencias
    public void ActivateEffect(Board board, ref bool[] mask, GameState state) {
        for (int i = 0; i < mask.Length; i++) {
            if (mask[i] && board[i].TryActivateEffect(state)) {
                board[i].ActivateEffect();
                mask[i] = false;
                state.Increase_PlayedCards();
            }
        }
    }
    2 referencias
    public void Attack(Board board, Board boardOpponent, ref bool[] mask, ref int life, GameState state) {
        for (int i = 0; i < mask.Length; i++) {
            if (boardOpponent.HaveSoldier()) for (int j = 1; j < boardOpponent.Length; j++) if (boardOpponent[j] is Soldier) if (TryAction.Attack(board, boardOpponent, i, j, mask)) {
                        Action.Attack(board, boardOpponent, i, j, ref mask);
                        state.Increase_PlayedCards();
                    }
            if (TryAction.Attack(board, boardOpponent, i, mask)) {
                Action.Attack(board, boardOpponent, i, ref mask);
                state.Increase_PlayedCards();
            }
            if (TryAction.Attack(board, i, mask)) {
                Action.Attack(board, i, ref mask, ref life);
                state.Increase_PlayedCards();
            }
```

La clase (la estrategia) **FullDefenseFullAttack** se caracteriza por una estrategia un tanto bruta, ya que a la hora de invocar prioriza la defensa y luego los soldados, y en activar los efectos depende si puede activarlos o no. En el combate primero tumba las cartas que pueden destruir su estructura, para luego centrarse en tumbar la estructura del oponente, y si luego puede seguir atacando, atacar a la vida del oponente (también, no tiene más nada que atacar como se ve en la imagen). Es una estrategia bruta, pero es medianamente eficaz, ya que, si puede activar más cartas con coste medianamente bajo, invoca la primera que encuentra en su recorrido, y en vez de ver que numero de cartas destruidas en su campo es mínimo en su campo y máximo en el campo del oponente, prefiere atacar al primero que se encuentre.

```
2 referencias
public static void TransformCardToText(Card card, string dir) {
    dir += @"\" + card.Name + ".txt";
    File.WriteAllText(dir, card.TransformCardToTXT());
}
1 referencia
public static Card TransformTextToCard(string dir) {
    List<string> list = MethNecesary.Extract(ReadTXT(dir));
    if (list[0] == TypeCards.Soldier.ToString()) {
        return new Soldier(list[1].Replace('_', ' '), int.Parse(list[2]), int.Parse(list[3]), int.Parse(list[4]), new Effect(list[5], int.Parse(list[6]), list[7], int.Parse(list[8])));
    }
    if (list[0] == TypeCards.Struct.ToString()) {
        return new Struct(list[1].Replace('_', ' '), int.Parse(list[2]), int.Parse(list[3]), new Effect(list[4], int.Parse(list[5]), list[6], int.Parse(list[7])));
    } throw new Exception();
}
1 referencia
private static string ReadTXT (string dir) {
    StreamReader sr = new StreamReader(dir);
    string line = sr.ReadToEnd();
    sr.Close();
    return line;
}
```

Finalmente veamos como una carta puede guardarse y como una carta puede cargarse.
Dentro de la clase **ReadFiles** tenemos 3 métodos que se tiene que ver primero.
El método *TransformTextToCard* cuyos parámetros tiene la carta y la dirección de la base de datos (lugar donde se guardan todas las cartas) efectúa el método que tiene la carta para transformar su contenido en string para luego tomar este y escribirlo en un .txt y el nombre de cada archivo es el nombre de la carta. Esto significa que si se quiere reescribir una carta basta con poner en el parámetro de name el nombre de la carta que se desea reescribir.
El método *TransformCardToText* y el método *ReadTXT* están relacionadas entre sí como se ve, la cual cada palabra del .txt se separa y se lee según la primera palabra de la lista, la cual es el tipo de carta (Soldier o Struct) para separarse y continuar con la lectura de cada palabra, y según la estructura de cada carta es como se crea y se devuelve. Ahora, si la primera palabra de la lista es diferente al nombre de la clase de alguna de las cartas heredadas de **Card** dará error código porque estas queriendo decir que existe otro tipo de carta diferente a los existentes.