Object Oriented Programming

Spring 2024

City Wars(Tokyo Reign)

Arman Yazdani

Phase 1:Intro

GitHub page & Trello board

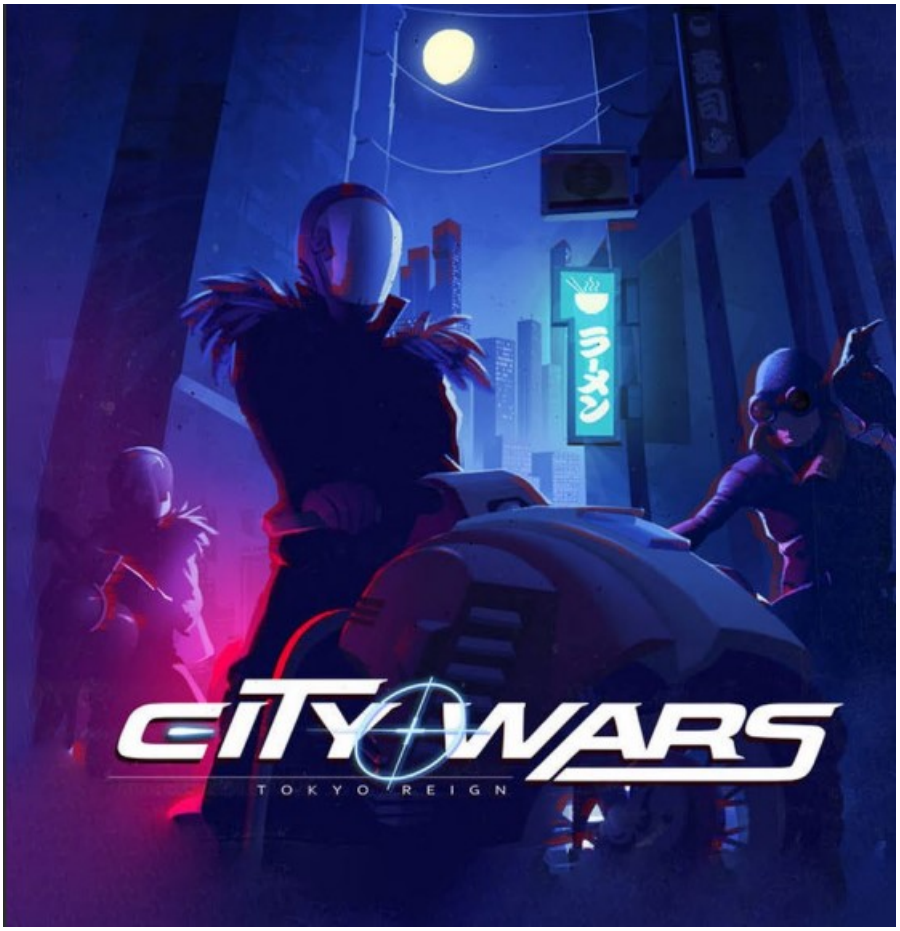


Figure 1: Game Cover

1. Time-Line Implementation
2. Cards&Spells
3. Classification
4. Data-Base
5. Upgrade&Reward&Punishment
6. Multi-player room
7. Single-player room

1 Time-Line Implementation

Figure 2: Time-Line

Sample Snippet code of implemented class

```java
public class Timeline {
    private List<Card> cardsOnTimeline;
    private int currentPosition;

    public Timeline() {
        cardsOnTimeline = new ArrayList<>();
        currentPosition = 0;
    }

    public void addCard(Card card) {
        cardsOnTimeline.add(card);
    }

    public void executeCurrentPosition() {
        for (Card card : cardsOnTimeline) {
            card.execute(currentPosition);
        }
        currentPosition++;
    }
}
```

In this class, we maintain a list of cards which are on the timeline. Then we define a method; addCard() with which we can add a card to the timeline. Then we can write another method; executeCurrentPosition() that executes all cards on the timeline. In this class, we have Card objects that are needed to have an 'execute(int position)' method that defines what happens when the card is executed at a given position on the timeline.

2 Cards&Spells

2.1 Damage/Healing Cards:

2.1.1 Ice Shard (Damage):

• Description: Launches sharp shards of ice at the enemy, causing frost damage.

• Ability: Inflicts 35 frost damage and slows the target's movement speed by 50% for 2 turns.

2.1.2 Regenerative Aura (Healing):

• Description: Emits a soothing aura, accelerating natural healing processes for allies.

• Ability: Restores 15 health per turn to all allies within range for 5 turns.

2.1.3 Searing Blaze (Damage):

• Description: Ignites the target with a blazing inferno, causing fire damage.

• Ability: Deals 50 fire damage to the target enemy.

2.1.4 Divine Shield (Healing):

• Description: Conjures a protective shield of divine energy, absorbing damage.

• Ability: Grants a shield that absorbs 60% of incoming damage for 2 turns.

2.1.5 Toxic Cloud (Damage):

• Description: Releases a noxious cloud of poison, damaging all enemies in the area.

• Ability: Deals 20 poison damage to all enemies within range per turn for 3 turns.

2.1.6 Life Drain (Healing):

• Description: Drains the life force from the enemy, transferring it to the caster.

• Ability: Absorbs 50% of the damage dealt to the target enemy and restores the caster's health by the same amount.

2.1.7 Fireball (Damage):

• Description: Launches a fiery projectile at the target, dealing moderate damage.

• Ability: Inflicts 50 damage to the target enemy.

2.1.8 Healing Aura (Healing):

• Description: Creates a soothing aura around allies, restoring their health.

• Ability: Heals all friendly units within range for 30 health.

2.1.9 Poisonous Dagger (Damage):

• Description: Inflicts a toxic wound on the target, causing damage over time.

• Ability: Deals 20 initial damage and an additional 10 damage per turn for 3 turns.

2.1.10 Regeneration (Healing):

• Description: Accelerates natural healing processes, gradually restoring health.

• Ability: Restores 10 health per turn for 5 turns to the caster.

2.1.11 Thunder Strike (Damage):

• Description: Calls forth a bolt of lightning to strike the enemy, dealing massive damage.

• Ability: Inflicts 100 damage to the target enemy.

2.1.12 Rejuvenating Spring (Healing):

• Description: Creates a spring of rejuvenating water, restoring vitality to allies.

• Ability: Heals all allies within range for 20 health points per turn for 3 turns.

2.1.13 Venomous Bite (Damage):

• Description: Injects a potent venom into the target, causing poison damage over time.

• Ability: Deals 30 initial damage and 15 poison damage per turn for 3 turns.

2.1.14 Holy Light(Healing):

• Description: Radiates healing energy, restoring health to allies with divine light.

• Ability: Heals the target ally for 50 health points.

2.1.15 Shadow Bolt (Damage):

• Description: Hurls a bolt of dark energy at the target, causing shadow damage.

• Ability: Inflicts 40 shadow damage to the target enemy.

2.2 Spell Cards:

2.2.1 Teleportation:

• Description: Instantly transports the caster to a designated location on the battlefield.

• Ability: Allows the caster to teleport to any visible location on the battlefield.

2.2.2 Barrier:

• Description: Creates a protective barrier around the caster, reducing incoming damage.

• Ability: Grants a shield that absorbs 50% of incoming damage for 3 turns.

2.2.3 Haste:

• Description: Temporarily enhances the caster's speed and agility, allowing for quick actions.

• Ability: Increases caster's movement speed and attack speed by 50% for 3 turns.

2.2.4 Silence:

• Description: Silences the target, preventing them from casting spells or using abilities.

• Ability: Silences the target for 2 turns, rendering them unable to cast spells or use abilities.

2.2.5 Meteor Shower:

• Description: Summons a rain of fiery meteors to bombard the battlefield, damaging all enemies.

• Ability: Deals 50 damage to all enemies on the battlefield.

3 Classification

3.1 Classes:

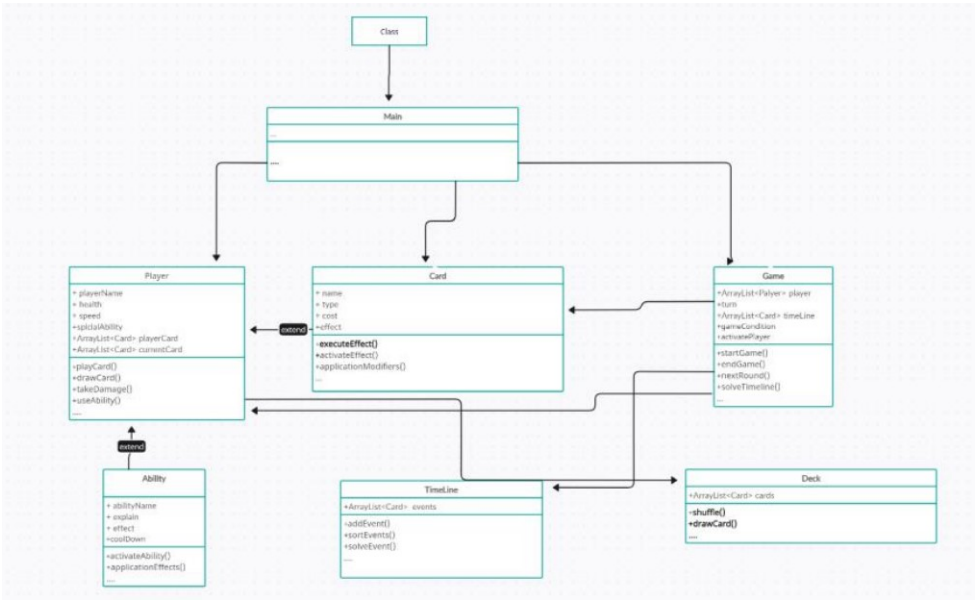


Figure 3: UML digram

3.1.1 Player Class:

```java
public class Player {
    private int health;
    private int mana;
    private List<Card> deck;
    private List<Card> hand;

    public void drawCard() {
        // Implement drawing a card from the deck
    }

    public void playCard(Card card) {
        // Implement playing a card from the hand
    }

    public void takeDamage(int damage) {
        // Implement taking damage
    }

    // Other methods...
}
```

• Properties: player name, health, speed, special abilities, cards in hand,cards available in the ground and so on.

• Methods: useAbility(),takeDamage(),drawCard() ,playCard(),etc.

• Relationships: Connect with Card class to interact with cards during the batlle.

3.1.2 Card Class:

```
public class Card {
    private String name;
    private String type;
    private int cost;
    // Other properties...

    public void executeEffect() {
        // Implement the effect of the card
    }

    // Other methods...
}
```

- Properties: Card name, type (attack, defense,special), effects, etc.
- Methods: activateEffect(),applicationModifiers(),etc.
- Relationships: Belongs to a Player class, interacts with the Game class for running.

3.1.3 Game Class:

```
public class Card {
    private String name;
    private String type;
    private int cost;
    // Other properties...

    public void executeEffect() {
        // Implement the effect of the card
    }

    // Other methods...
}
```

- Properties: Players list, game status, current round, active player, timeline, etc.
- Methods: solveTimeline(),nextRound(),endGame() ,startGame(),etc.
- Relationships: Manages interactions between players the timeline and card actions.

3.1.4 TimeLine Class:

```
public class Card {
    private String name;
        private String type;
        private int cost;
        // Other properties...

        public void executeEffect() {
        // Implement the effect of the card
    }

// Other methods...
}
```

- Properties: Players list, game status, current round, active player, timeline, etc.
- Methods: solveTimeline(),nextRound(),endGame() ,startGame(),etc.
- Relationships: Manages interactions between players the timeline and card actions.

3.1.5 Ability Class (Optional):

- Properties: Ability names, descriptions, effects, cooldown, etc.
- Methods: Activision(),aplication(), etc.
- Relationships: To use special abilities is linked to the Player class.

3.2 Main Structure:

- Game Class:

Gameplay coordination player interaction management, card play, and timeline settlement.

- Players:

They interact with the cards through the Player class. They draw cards, play cards, use special abilities and perform actions.

- Card Class:

Defines the behavior of each card, including how it affects players and the game state when activated.

- TimeLine Class:

Manages the sequence of events in the game and ensures that card actions are executed in the correct order.

- Ability Class:

Grants players unique skills or powers that can affect gameplay and strategy

4 Data-Base

For saving information of card upgrades and ¡ in a database we need to set up a database schema to store the information.
For example for storing information of a player including it's name or id and¡ we can structure this table:

Column Name	Data Type	Description
player_id	INT	Primary key, unique identifier for each player
name	VARCHAR	Name of the player
...	...	Other player attributes

Or for storing information of each card in the game we can struct this table:

Column Name	Data Type	Description
card_id	INT	Primary key, unique identifier for each card
name	VARCHAR	Name of the card
description	TEXT	Description of the card
type	VARCHAR	Type of the card(e.g.,Damage,Healing,Spell)
...	...	Other card attributes

With this schema we can do the following operations to manage information:

- **Create:** When a player obtains a new card or upgrades an existing card, insert a new row into the PlayerCards table with the player's ID, the card's ID, and any upgrade information.

- **Read:** Retrieve player information, card information, or player-card relationships as needed to display in the game interface.
- **Update:** Modify player information, card information, or player-card relationships when players upgrade their cards or perform other actions that affect their data.
- **Delete:** Remove player information, card information, or player-card relationships when players delete their accounts or perform other actions that require data removal.

5 Upgrade&Reward&Punishment

Players can buy their cards by spending money (coins or diamonds) or in-game resources that can be used with Play games to earn, upgrade. The exact mechanism of upgrading cards can be different depending on the game design. a typical idea would be:

```
public class Player {
    private int currency;
    // Other properties...
    public void upgradeCard(Card card) {
        if (currency >= card.getUpgradeCost()) {
            currency -= card.getUpgradeCost();
            card.upgrade();}
        else {
            System.out.println("Not enough currency to upgrade the card."
               );}
    }
    // Other methods...
}
```

• In this example, each Card object has an upgradeCost property and the Player object has a currency property. When the player wants to upgrade a card, he calls the upgrade method of the card which checks whether it has enough currency or not. If they do, it deducts the fee from their currency and calls the upgrade method on the card.

• As for rewards and penalties, these also depend on game design. A common approach is that Reward the winner with in-game money or new cards and give the loser a smaller amount of currency (or may not add money).

Here, for example, the winner is given twice the currency of the loser:

```
public class Game {
    // Other properties...
    public void endGame(Player winner, Player loser) {
        winner.addCurrency(100); // Reward the winner with 100 currency
        loser.addCurrency(50); // Give the loser 50 currency
    }
    // Other methods...
}
```

6 Multi-player room

• To handle player identification and account creation we represent a class for individual players that define some properties as username and password:

```
public class Player {
    private String username;
    private String password;
    // Other player attributes (e.g., level, experience, currency)
```

```
    public Player(String username, String password) {
        this.username = username;
        this.password = password;
        // Initialize other attributes
    }
    // Getter and setter methods for player attributes
    public boolean authenticate(String inputPassword) {
        // Check if the provided password matches the player's password
        return password.equals(inputPassword);
    }
}
```

• Then we define an AccountManager Class to store player data and create new accounts. This class manages player's account by storing them in a map (username as key and Player object as value). It provides methods to create new accounts and authenticate players.

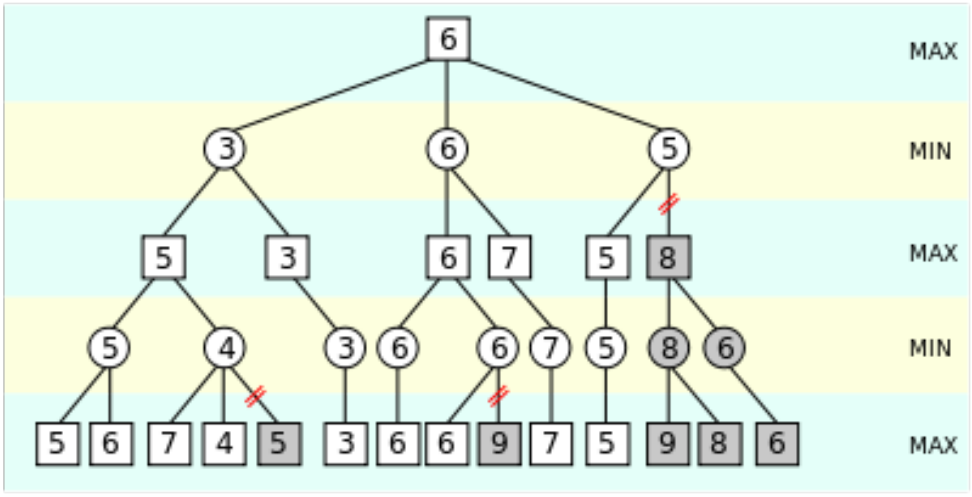
```
public class AccountManager {
    private Map<String,Player> players;
    public AccountManager() {
        this.players = new HashMap<>();}
    public void createAccount(String username, String password) {
        // Check if the username is available
        if (!players.containsKey(username)) {
            // Create a new player object and add it to the players map
            players.put(username, new Player(username, password));
            System.out.println("Account created successfully for username
               : " + username);}
        else {
            System.out.println("Username already exists. Please choose
               another username.");
        }
    }
    public Player getPlayer(String username) {
        // Retrieve player object by username
        return players.get(username);
    }
}
```

7 Single-player room

If we run the game as a single-player game with an AI opponent, we need to create an AI algorithm that can make decisions and effectively work against the player. play cards.

There are some common algorithms and strategies that can be considered to create an artificial intelligence opponent in the game "Tokyo Region War City",such as *Minimax Algorithm with Alpha-Beta Pruning*,*Monte Carlo Tree Search (MCTS)*,*Rule-Based System*, *Genetic Algorithm*,etc.

Let's have a closer look on Minimax Algorithm with Alpha-Beta Pruning:



• This algorithm is a decision-making algorithm that is usually used in two-player games. It evaluates the possible moves of both players to determine the best move for the AI slow.

• Alpha-beta pruning is used to reduce the number of evaluated nodes in the search tree and make the algorithm more efficient.

◁ Let's break down the steps and provide an example.

- **Minimax Algorithm:** The minimax algorithm is used for decision-making in two-player games. It explores the game tree by recursively evaluating possible moves and choosing the best move for the maximizing player (usually the AI) and the worst move for the minimizing player (usually the opponent).
- **Alpha-Beta Pruning:** Alpha-beta pruning is an optimization technique that reduces the number of nodes evaluated in the minimax search. It maintains two values, alpha (the best value for the maximizing player) and beta (the best value for the minimizing player). If a node's value falls outside the alpha-beta window, it can be pruned (skipped).
- Sample Java code:

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):
if node is a leaf node:
return value of the node
if isMaximizingPlayer:
bestVal = -INFINITY
for each child node:
value = minimax(node, depth + 1, false, alpha, beta)
bestVal = max(bestVal, value)
alpha = max(alpha, bestVal)
if beta <= alpha:
break
return bestVal
else:
bestVal = +INFINITY
for each child node:
value = minimax(node, depth + 1, true, alpha, beta)
bestVal = min(bestVal, value)
beta = min(beta, bestVal)
if beta <= alpha:
break
return bestVal
// Calling the function for the first time
minimax(rootNode, 0, true, -INFINITY, +INFINITY)
```