Uno Game

Rawan Afaneh

What are we required to do in this assignment:

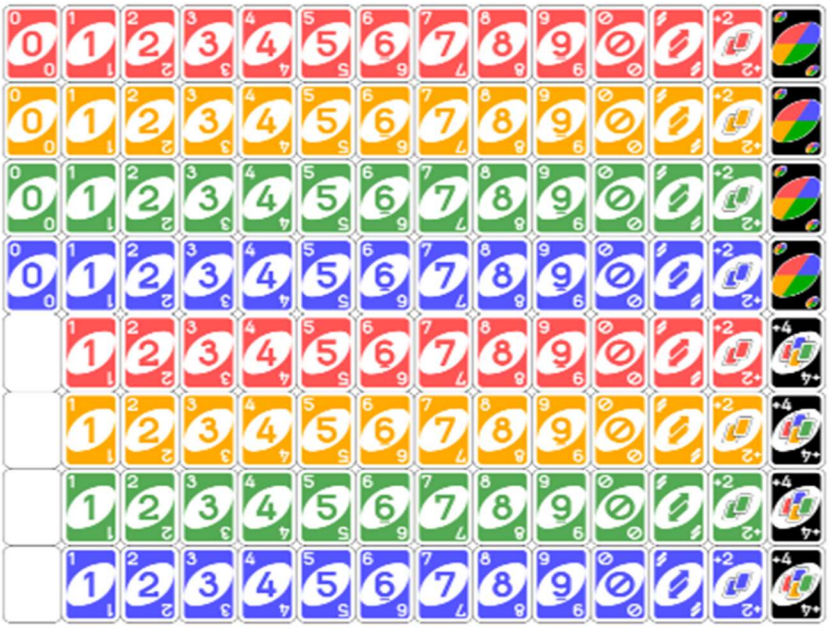
Uno Game Engine

In this assignment, we are required to build an Uno game engine to be used by other developers (not players).

First, let us go over some background about the game. Uno is a card game that is typically played by at 2-10 players, and contains 108 cards divided as follows:

- Numbered cards (0-9)
- Action cards (Reverse, Skip, Draw Two)
- Wild cards (Wild, Wild Draw Four)

This is a picture that shows the details of the 108 cards.



Let me begin by presenting the classes and their respective functionalities:

1- GameDriver:

```java
public class GameDriver {
    public static void main(String[] args) throws Exception {
        Game game = new UnoGameVariation();
        game.play();
    }
}
```

The **GameDriver** class serves as the entry point for running the Uno game. Its main responsibility is to create an instance of the **UnoGameVariation** class and start the game by calling the **play** method.
Responsibilities:

- Declare a public class named **GameDriver**.
- Inside the **main** method:
    - Create an instance of the **UnoGameVariation** class and assign it to the **game** variable of type **Game**.
    - Call the **play** method on the **game** object to start the Uno game.

2- Game:

```java
import java.util.*;
2 usages  1 inheritor
public abstract class Game {
    3 usages
    private final List<Player> players;
    1 usage
    public Game() {
        players = createPlayers();
    }
    1 usage  1 implementation
    protected abstract List<Player> createPlayers();
    2 usages  1 implementation
    protected abstract void initializeGame(List<Player> players);
    2 usages  1 implementation
    protected abstract boolean isGameEnded();
    7 usages  1 implementation
    protected abstract void processTurn(Player player);
    1 usage  1 implementation
    protected abstract void processActionCard(UnoCard card, Player player);
    1 usage  1 implementation
    protected abstract void processNumberCard(UnoCard card, Player player);
    1 usage  1 implementation
    protected abstract void processWildCard(UnoCard card, Player player);
    2 usages  1 implementation
    protected abstract void endGame() ;
    1 usage
    public void play() {
        initializeGame(getPlayers());
        processTurn(players.get(0));
    }
    19 usages
    public List<Player> getPlayers() {
        return players;
    }

}
```

The **Game** class is an abstract class that serves as the base class for different game variations. It defines the common structure and behavior of a game. Subclasses are expected to provide concrete implementations for certain methods specific to their respective game variations.
Responsibilities:

- Declare a public abstract class named **Game**.
- Declare a private final instance variable **players** of type **List<Player>** to store the players participating in the game.
- In the constructor **Game()**, initialize the **players** list by calling the abstract method **createPlayers()**.
- Declare the following abstract methods that must be implemented by subclasses:
    - **createPlayers()**: Creates and returns a list of players for the game.
    - **initializeGame(List<Player> players)**: Initializes the game, such as distributing cards or setting up the game board.
    - **isGameEnded()**: Checks if the game has ended.

- **processTurn(Player player)**: Processes a turn for the specified player.
- **processActionCard(UnoCard card, Player player)**: Processes an action card played by the player.
- **processNumberCard(UnoCard card, Player player)**: Processes a number card played by the player.
- **processWildCard(UnoCard card, Player player)**: Processes a wild card played by the player.
- **endGame()**: Handles the end of the game.

3- CardColor:

```
public enum CardColor {
    1 usage
    RED,
    1 usage
    BLUE,
    1 usage
    GREEN,
    1 usage
    YELLOW,
    4 usages
    NONE
}
```

The **CardColor** enum represents the different colors that can be associated with a card in a game. It provides a predefined set of color options.
The enum constants defined in **CardColor** are:

- **RED**: Represents the color red.
- **BLUE**: Represents the color blue.
- **GREEN**: Represents the color green.
- **YELLOW**: Represents the color yellow.
- **NONE**: Represents a special value indicating no specific color.

4- CardType:

```
public enum CardType {
    1 usage
    ZERO,
    no usages
    ONE,
    no usages
    TWO,
    no usages
    THREE,
    no usages
    FOUR,
    no usages
    FIVE,
    no usages
    SIX,
    no usages
    SEVEN,
    no usages
    EIGHT,
    no usages
    NINE,
    3 usages
    REVERSE,
    3 usages
    SKIP,
    3 usages
    DRAW_TWO,
    2 usages
    WILD,
    4 usages
    WILD_DRAW_FOUR
}
```

The **CardType** enum represents the different types of cards that can be used in a game. It provides a predefined set of card types.
The enum constants defined in **CardType** are:

- **ZERO** to **NINE**: Represents the number cards from 0 to 9.
- **REVERSE**: Represents the reverse card.
- **SKIP**: Represents the skip card.
- **DRAW_TWO**: Represents the draw two card.
- **WILD**: Represents the wild card.
- **WILD_DRAW_FOUR**: Represents the wild draw four card.

5- UnoCard:

```
public class UnoCard {
    4 usages
    private CardType type;
    3 usages
    private CardColor color;
    2 usages
    private int number = -1;
    public UnoCard(CardType type, CardColor color, int number) {
        this.type = type;
        this.color = color;
        this.number = number;
    }
    public UnoCard(CardType type, CardColor color) {
        this.type = type;
        this.color = color;

    }
    public UnoCard(CardType type) {
        this.type = type;

    }
    11 usages
    public CardType getType() { return type; }

    12 usages
    public CardColor getColor() { return color; }
    3 usages
    public int getNumber() { return number; }
}
```

The **UnoCard** class represents a card in the game of Uno. It has three private member variables: **type**, **color**, and **number**.
The constructors of the **UnoCard** class allow for creating different types of Uno cards:

- **UnoCard(CardType type, CardColor color, int number)**: Creates a card with the specified type, color, and number. This constructor is used for number cards.
- **UnoCard(CardType type, CardColor color)**: Creates a card with the specified type and color. This constructor is used for action cards.
- **UnoCard(CardType type)**: Creates a card with only the specified type. This constructor is used for wild cards.
  The **getType()** method returns the type of the card (**CardType** enum). The **getColor()** method returns the color of the card (**CardColor** enum). The **getNumber()** method returns the number of the card (applicable to number cards only).

6- CardHandler:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

8 usages
public class CardHandler {
    4 usages
    public static UnoCard drawCardFromDeck(List<UnoCard> unoCards) {

        int randomIndex = (int) (Math.random() * unoCards.size());
        System.out.println("the cards number is: " + unoCards.size());
        System.out.println("Draw another card...");
        UnoCard card = unoCards.remove(randomIndex);
        Scanner sc = new Scanner(System.in);
        sc.nextLine();
        return card;
    }

    1 usage
    public static List<UnoCard> getValidMoves(List<UnoCard> hand, UnoCard lastCard, CardColor color) {
        List<UnoCard> validMoves = new ArrayList<>();
        for (UnoCard c : hand) {
            if (isValidMove(c, color, lastCard)) {
                validMoves.add(c);
            }
        }
        return validMoves;
    }

    2 usages
    public static boolean isValidMove(UnoCard card, CardColor color, UnoCard lastCard) {
        if (checkIfNumbers(card) || checkIfAction(card)) {
            if (color == CardColor.NONE) {
                if (card.getColor() == CardColor.BLUE || card.getColor() == CardColor.GREEN ||
                        card.getColor() == CardColor.RED || card.getColor() == CardColor.YELLOW ||
                        card.getNumber() == lastCard.getNumber()) {
                    return true;
                }
            } else {
                if (color.equals(card.getColor()) || card.getType() == lastCard.getType()) {
                    return true;
                }
            }
        } else if (checkIfWild(card)) {
            return true;
        }
        return false;
    }

    2 usages
    public static boolean checkIfNumbers(UnoCard card) { return card.getNumber() != -1; }

    2 usages
    public static boolean checkIfAction(UnoCard card) {
        CardType cardType = card.getType();
        return cardType == CardType.SKIP || cardType == CardType.REVERSE || cardType == CardType.DRAW_TWO;
    }

    2 usages
    public static boolean checkIfWild(UnoCard card) {
        CardType cardType = card.getType();
        return cardType == CardType.WILD_DRAW_FOUR || cardType == CardType.WILD;
    }

    1 usage
    public static void displayAvailableMoves(List<UnoCard> validMoves) {
        System.out.println();
        System.out.println("Available moves: ");
        for (int i = 0; i < validMoves.size(); i++) {
            System.out.println((i + 1) + ". " + validMoves.get(i).getType() + " " + validMoves.get(i).getColor());
        }
    }

    1 usage
    public static UnoCard handleNoValidMoves(Player player, CardColor color, UnoCard lastCard, List<UnoCard> unoCards) {
        System.out.println();
        System.out.println("No valid moves. Drawing a card...");
        UnoCard drawnCard = drawCardFromDeck(unoCards);
        player.addCardToHand(drawnCard);
        System.out.println("The card is: " + drawnCard.getType() + " - " + drawnCard.getColor());

        while (!isValidMove(drawnCard, color, lastCard)) {
            drawnCard = drawCardFromDeck(unoCards);
            player.addCardToHand(drawnCard);
            System.out.println("The card is: " + drawnCard.getType() + " - " + drawnCard.getColor());
        }
        return drawnCard;

    }
}
```

The **CardHandler** class contains several static methods for handling Uno cards and performing card-related operations.

- **drawCardFromDeck(List<UnoCard> unoCards)**: This method randomly selects a card from the provided deck (**unoCards**) and removes it from the deck. It waits for user input before returning the drawn card.
- **getValidMoves(List<UnoCard> hand, UnoCard lastCard, CardColor color)**: This method returns a list of valid moves from the given **hand** of cards, based on the **lastCard** played and the current **color** in play.
- **isValidMove(UnoCard card, CardColor color, UnoCard lastCard)**: This method checks if a specific **card** is a valid move based on the **color**, **lastCard**, and the rules of Uno. It considers number cards, action cards, and wild cards.
- **checkIfNumbers(UnoCard card)**: This method checks if a card is a number card by checking its **number** property.
- **checkIfAction(UnoCard card)**: This method checks if a card is an action card (SKIP, REVERSE, or DRAW_TWO) by checking its **type** property.
- **checkIfWild(UnoCard card)**: This method checks if a card is a wild card (WILD or WILD_DRAW_FOUR) by checking its **type** property.
- **displayAvailableMoves(List<UnoCard> validMoves)**: This method displays the available moves (cards) from the given list of **validMoves** to the console.
- **handleNoValidMoves(Player player, CardColor color, UnoCard lastCard, List<UnoCard> unoCards)**: This method is called when a player has no valid moves. It draws a card from the deck (**unoCards**) and adds it to the player's hand. If the drawn card is still not a valid move, it continues drawing cards until a valid move is obtained. The drawn card is returned.

7- Player:

```java
import java.util.*;
52 usages
public class Player {
    2 usages
    private final String name;
    5 usages
    private List<UnoCard> hand;
    2 usages
    private int score;
    1 usage
    public Player(String name) {
        this.name = name;
        hand = new ArrayList<>();
    }
    10 usages
    public List<UnoCard> getHand() { return hand; }
    1 usage
    public void setHand(List<UnoCard> hand) { this.hand = hand; }
    3 usages
    public int getScore() { return score; }
    1 usage
    public void setScore(int score) { this.score = score; }
    12 usages
    public String getName() { return name; }
    4 usages
    public void addCardToHand(UnoCard drawnCard) {
        hand.add(drawnCard);
    }
    3 usages
    public void removeCardfromHand(UnoCard removeCard) { hand.remove(removeCard); }
}
```

The **Player** class represents a player in the Uno game. It has the following properties and methods:

- Properties:
    - **name**: A string representing the name of the player.
    - **hand**: A list of **UnoCard** objects representing the player's current hand of cards.
    - **score**: An integer representing the player's score.
- Constructor:
    - **Player(String name)**: Initializes a new player with the given name. It sets the **name** property and initializes an empty **hand** list.
- Methods:
    - **getHand()**: Returns the player's current hand of cards.
    - **setHand(List<UnoCard> hand)**: Sets the player's hand to the specified list of cards.
    - **getScore()**: Returns the player's current score.
    - **setScore(int score)**: Sets the player's score to the specified value.
    - **getName()**: Returns the name of the player.
    - **addCardToHand(UnoCard drawnCard)**: Adds the given **drawnCard** to the player's hand.
    - **removeCardFromHand(UnoCard removeCard)**: Removes the specified **removeCard** from the player's hand.

8- PlayerInputHandler:

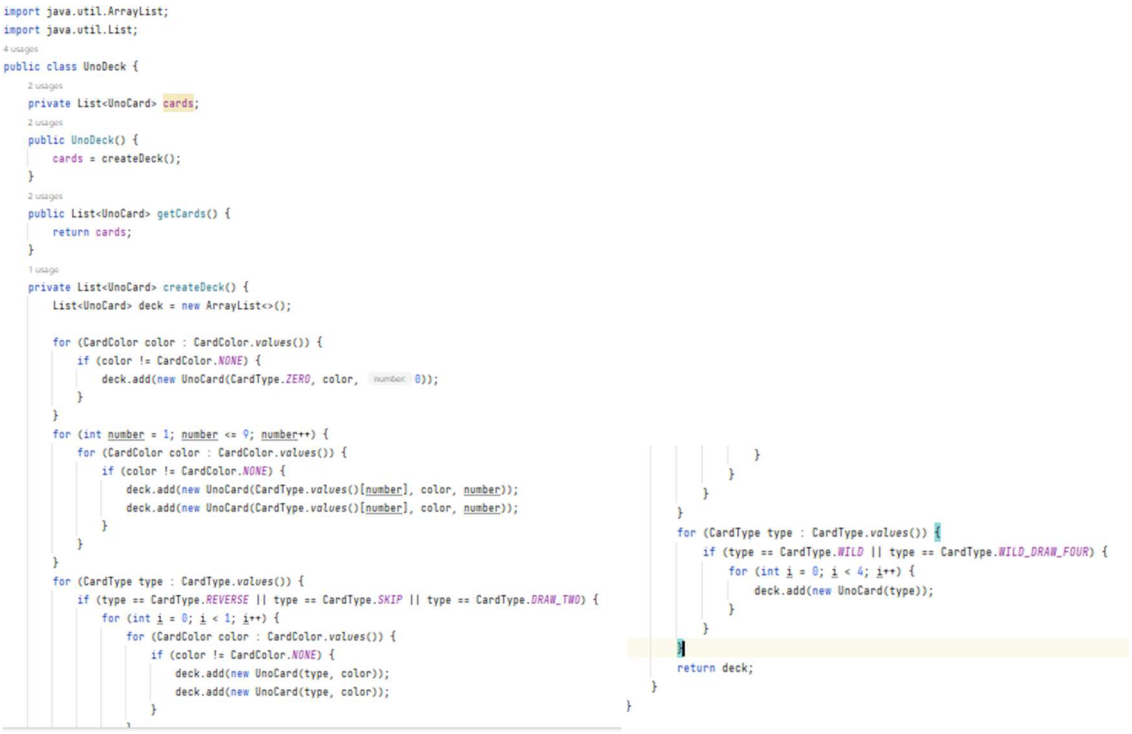
```java
import java.util.*;
12 usages
public class PlayerInputHandler {
    1 usage
    public static UnoCard getPlayerMove(Player player) {
        System.out.println("Choose a move");
        Scanner scanner = new Scanner(System.in);
        int moveIndex;
        UnoCard card;
        do {
            moveIndex = scanner.nextInt();
            if (moveIndex < 1 || moveIndex > player.getHand().size()) {
                System.out.println("Invalid move. Try again.");
            }
        } while (moveIndex < 1 || moveIndex > player.getHand().size());

        card = player.getHand().get(moveIndex - 1);
        return card;
    }
    1 usage
    public static UnoCard getPlayerMove(Player player, List<UnoCard> validMoves) {
        System.out.println("Choose a move (1-" + validMoves.size() + "):");
        Scanner scanner = new Scanner(System.in);
        int moveIndex;
        UnoCard card;
        do {
            moveIndex = scanner.nextInt();
            if (moveIndex < 1 || moveIndex > validMoves.size()) {
                System.out.println("Invalid move. Try again.");
            }
        } while (moveIndex < 1 || moveIndex > validMoves.size());
        card = validMoves.get(moveIndex - 1);
        return card;
    }
    9 usages
    public static Player getNextPlayer(Player currentPlayer, List<Player> player) {
        int currentPlayerIndex = player.indexOf(currentPlayer);
        int nextPlayerIndex = (currentPlayerIndex + 1) % player.size();
        return player.get(nextPlayerIndex);
    }
    1 usage
    public static List<Player> getPlayerOrder(List<Player> players) {
        List<Player> playerOrder = new ArrayList<>();
        System.out.println("Player order:");
        for (int i = 0; i < players.size(); i++) {
            Player player = players.get(i);
            System.out.println((i + 1) + ". " + player.getName());
            playerOrder.add(player);
        }
        return playerOrder;
    }
}
```

Description: The **PlayerInputHandler** class is responsible for handling player input and interactions during the Uno game. It provides methods to retrieve and process player input for various game actions. Here is a detailed description of its responsibilities:

1. **getPlayerMove(Player player)**: This method prompts the specified player to choose a move and reads their input from the console. It validates the input to ensure it corresponds to a valid move in the player's hand. It returns the chosen Uno card.
2. **getPlayerMove(Player player, List<UnoCard> validMoves)**: This overloaded method is used when there are valid moves available for the player. It prompts the player to choose a move from the list of valid moves and reads their input from the console. It validates the input and returns the chosen Uno card.
3. **getNextPlayer(Player currentPlayer, List<Player> players)**: This method determines the next player in the player list based on the current player's index. It uses modular arithmetic to handle looping back to the first player when reaching the end of the list. It returns the next player.
4. **getPlayerOrder(List<Player> players)**: This method establishes and displays the player order for the current game. It prints the order of players with their corresponding numbers and returns a list representing the player order.

Overall, the **PlayerInputHandler** class facilitates the interaction between the game mechanics and player input. It ensures that player moves are captured correctly, validates input, determines the next player, and manages the player order during gameplay.

9- UnoDeck:



```
import java.util.ArrayList;
import java.util.List;
4 usages
public class UnoDeck {
    2 usages
    private List<UnoCard> cards;
    2 usages
    public UnoDeck() {
        cards = createDeck();
    }
    2 usages
    public List<UnoCard> getCards() {
        return cards;
    }
    1 usage
    private List<UnoCard> createDeck() {
        List<UnoCard> deck = new ArrayList<>();

        for (CardColor color : CardColor.values()) {
            if (color != CardColor.NONE) {
                deck.add(new UnoCard(CardType.ZERO, color,  number: 0));
            }
        }
        for (int number = 1; number <= 9; number++) {
            for (CardColor color : CardColor.values()) {
                if (color != CardColor.NONE) {
                    deck.add(new UnoCard(CardType.values()[number], color, number));
                    deck.add(new UnoCard(CardType.values()[number], color, number));
                }
            }
        }
        for (CardType type : CardType.values()) {
            if (type == CardType.REVERSE || type == CardType.SKIP || type == CardType.DRAW_TWO) {
                for (int i = 0; i < 1; i++) {
                    for (CardColor color : CardColor.values()) {
                        if (color != CardColor.NONE) {
                            deck.add(new UnoCard(type, color));
                            deck.add(new UnoCard(type, color));
                        }
                    }
                }
            }
        }
        for (CardType type : CardType.values()) {
            if (type == CardType.WILD || type == CardType.WILD_DRAW_FOUR) {
                for (int i = 0; i < 4; i++) {
                    deck.add(new UnoCard(type));
                }
            }
        }
        return deck;
    }
}
```

Description: The **UnoDeck** class represents a deck of Uno cards. It is responsible for creating and managing the cards in the deck.

Responsibilities:

1. Creating the Deck: The **createDeck()** method is a private helper method that generates a list of Uno cards representing a complete deck. It iterates through the **CardColor** enum to create cards of each color, including zero cards. It also adds number cards (1-9) for each color, action cards (Reverse, Skip, Draw Two) for each color, and wild cards (Wild, Wild Draw Four). The created cards are added to the deck list.
2. Initializing the Deck: The constructor **UnoDeck()** initializes the **cards** list by calling the **createDeck()** method, which populates the list with the initial set of Uno cards.
3. Accessing the Cards: The **getCards()** method allows external classes to access the list of Uno cards in the deck. It returns the **cards** list.

10- UnoGameVariation:

```
import java.util.*;
public class UnoGameVariation extends Game {
    private UnoCard lastCard;
    private CardColor color = CardColor.valueOf("NONE");
    UnoDeck unoDeck = new UnoDeck();
    List<UnoCard> unoCards = unoDeck.getCards();
    @Override
    protected List<Player> createPlayers() {
        List<Player> players = new ArrayList<>();
        System.out.println("Please enter the number of players:");
        Scanner scanner = new Scanner(System.in);
        int number = 0;
        try {
            number = scanner.nextInt();
            while (number < 2 || number > 10) {
                System.out.println("The game is typically played
by 2-10 players.");
                number = scanner.nextInt();
            }
        } catch (InputMismatchException e) {
            System.out.println("Invalid input. Please enter a
number.");
            scanner.nextLine();
        }
        scanner.nextLine();
        for (int i = 0; i < number; i++) {
            System.out.println("Please enter the name of player "
+ (i + 1) + ":");
            String name = scanner.nextLine();
            players.add(new Player(name));
        }
        System.out.println("Player order:");
        for (int i = 0; i < players.size(); i++) {
            System.out.println((i + 1) + ". " +
players.get(i).getName());
        }
        return players;
    }
    @Override
    protected void initializeGame(List<Player> players) {
        for (Player player : players) {
            List<UnoCard> hand = new ArrayList<>();
            for (int j = 0; j < 7; j++) {
                int randomIndex = (int) (Math.random() *
unoCards.size());
                UnoCard card = unoCards.remove(randomIndex);
                hand.add(card);
            }
            player.setHand(hand);
        }
        System.out.println("----------------------------");
        System.out.println("Starting Uno game!");
        System.out.println("----------------------------");
    }
    @Override
    protected void processTurn(Player player) {
        System.out.println("-----------------------------");
        System.out.println("It's " + player.getName() + "'s
turn.");
        if (lastCard != null) {
            System.out.println("last card: " + lastCard.getType()
+ " - " + lastCard.getColor());
        }
        System.out.println();
        System.out.println("Your cards are: ");
        for (int i = 0; i < player.getHand().size(); i++)
            System.out.println((i + 1) + ": " +
player.getHand().get(i).getType() + " - " +
player.getHand().get(i).getColor());
        UnoCard card = null;
        if (color == CardColor.valueOf("NONE")) {
            card = PlayerInputHandler.getPlayerMove(player);
        } else {
            List<UnoCard> hand = player.getHand();
            List<UnoCard> validMoves =
CardHandler.getValidMoves(hand, lastCard, color);

            if (validMoves.isEmpty()) {
                card = CardHandler.handleNoValidMoves(player,
color, lastCard, unoCards);
            } else {
                CardHandler.displayAvailableMoves(validMoves);
                card = PlayerInputHandler.getPlayerMove(player,
validMoves);
            }
        }
        if (CardHandler.checkIfNumbers(card)) {
            processNumberCard(card, player);
        } else if (CardHandler.checkIfAction(card)) {
            processActionCard(card, player);
        } else if (CardHandler.checkIfWild(card)) {
            processWildCard(card, player);
        } else {
            System.out.println("Invalid move. Try again.");
            processTurn(player);
        }
    }
```

```
@Override
    protected void processNumberCard(UnoCard card, Player player) {
        lastCard = card;
        color = card.getColor();
        player.removeCardfromHand(card);
        if (isGameEnded()) {
            endGame();
        } else processTurn(PlayerInputHandler.getNextPlayer(player,
getPlayers()));
    }
    @Override
    protected void processActionCard(UnoCard card, Player player) {
        lastCard = card;
        color = card.getColor();
        player.removeCardfromHand(card);

        Player p = CardFactory.actionCards(player, getPlayers(),
card.getType(), unoCards);
        processTurn(p);
    }
    @Override
    protected void processWildCard(UnoCard card, Player player) {
        lastCard = card;
        Scanner scanner = new Scanner(System.in);
        CardType cardType = card.getType();
        Player nextPlayer = PlayerInputHandler.getNextPlayer(player,
getPlayers());
        if (cardType == CardType.WILD_DRAW_FOUR) {
            System.out.println(nextPlayer.getName() + " must draw
four cards!");
            for (int i = 0; i < 4; i++) {
                UnoCard drawnCard =
CardHandler.drawCardFromDeck(unoCards);
                nextPlayer.addCardToHand(drawnCard);
            }
            System.out.println(nextPlayer.getName() + "'s turn is
skipped!");
        }
        System.out.println("Wild Card! Choose a color (Red, Blue,
Green, Yellow):");
        String chosenColor = scanner.nextLine().toLowerCase();
        while (!chosenColor.matches("red|blue|green|yellow")) {
            System.out.println("Invalid color. Choose a color (Red,
Blue, Green, Yellow):");
            chosenColor = scanner.nextLine().toLowerCase();
        }
        color = CardColor.valueOf(chosenColor.toUpperCase());
        player.removeCardfromHand(card);
        if (isGameEnded()) {
            endGame();
        } else {
            if (cardType == CardType.WILD_DRAW_FOUR) {

processTurn(PlayerInputHandler.getNextPlayer(nextPlayer,
getPlayers()));
            } else {
                processTurn(PlayerInputHandler.getNextPlayer(player,
getPlayers()));
            }
        }
    }
    @Override
    protected void endGame() {
        for (int i = 0; i < getPlayers().size(); i++) {
            if (getPlayers().get(i).getHand().size() == 0) {

getPlayers().get(i).setScore(getPlayers().get(i).getScore() + 100);
            }
        }
        for (int i = 0; i < getPlayers().size(); i++) {
            System.out.println(getPlayers().get(i).getName() + "'s
score is :" + getPlayers().get(i).getScore());
            if (getPlayers().get(i).getScore() == 500) {

System.out.println(getPlayers().get(i).getName().toUpperCase() + " is
THE WINNER !! ");
                System.exit(0);
            }

        }
        System.out.println("The round finished! ");
        System.out.println("-------------------------");
        for (Player player : getPlayers()) {
            player.getHand().clear();
        }
        System.out.println("Starting a new round!");
        Scanner sc = new Scanner(System.in);
        sc.nextLine();
        lastCard = null;
        color = CardColor.valueOf("NONE");
        UnoDeck unoDeck = new UnoDeck();
        unoCards = unoDeck.getCards();
        initializeGame(getPlayers());
        processTurn(getPlayers().get(0));
    }
    @Override
    protected boolean isGameEnded() {
        for (Player p : getPlayers()) {
            if (p.getHand().size() == 0) return true;
        }
        return false;
    }
}
```

Description: The **UnoGameVariation** class represents a variation of the Uno game. It extends the **Game** class, which suggests that it inherits and overrides certain game-related methods. This class incorporates the mechanics and rules specific to the Uno game variation.

Responsibilities:

1. Creating Players: The **createPlayers()** method prompts the user to enter the number of players and their names. It creates a list of **Player** objects based on the user input and returns it.
2. Initializing the Game: The **initializeGame(List<Player> players)** method initializes the game by assigning a random set of Uno cards to each player's hand. It uses the **UnoDeck** class to create a deck of Uno cards. Each player is dealt 7 cards.
3. Processing a Turn: The **processTurn(Player player)** method handles a player's turn in the game. It displays the current player's turn, the last played card, and their hand. It prompts the player to choose a move and processes the chosen move based on the type of card. It calls appropriate methods to handle number cards, action cards, and wild cards.
4. Processing Number Cards: The **processNumberCard(UnoCard card, Player player)** method updates the last played card, removes the played card from the player's hand, and sets the new color if necessary. It then proceeds to the next player's turn unless the game has ended.
5. Processing Action Cards: The **processActionCard(UnoCard card, Player player)** method handles action cards such as Reverse, Skip, and Draw Two. It updates the last played card, removes the played card from the player's hand, and calls the **CardFactory.actionCards()** method to handle the corresponding action.
6. Processing Wild Cards: The **processWildCard(UnoCard card, Player player)** method handles wild cards and wild draw four cards. It prompts the player to choose a color and sets it as the new color. For wild draw four cards, it forces the next player to draw four cards and skip their turn. It then proceeds to the next player's turn unless the game has ended.
7. Ending the Game: The **endGame()** method is called when the game has ended. It calculates and displays the scores of each player, checks if any player has reached a score of 500 (indicating they have won the game), and starts a new round by clearing the players' hands and initializing a new deck of Uno cards.
8. Checking if the Game Has Ended: The **isGameEnded()** method checks if any player's hand is empty, indicating that the game has ended. It returns **true** if the game has ended and **false** otherwise.

11- CardFactory:

```
import java.util.List;

1 usage
public class CardFactory {

    1 usage
    public static Player actionCards(Player player, List<Player> players, CardType type, List<UnoCard> unoCards) {
        switch (type) {
            case SKIP:
                return SkipCard.skipCard(player, players);
            case REVERSE:
                return ReverseCard.reverseCard(players, player);
            case DRAW_TWO:
                return DrawTwoCard.drawTwoCard(player, players, unoCards);
            default:
                throw new IllegalArgumentException("Invalid card type");
        }
    }

}
```

Description: The **CardFactory** class is responsible for handling action cards in the Uno game. It provides a static method to process the effects of different action cards on players.

Responsibilities:

1. Processing Action Cards: The **actionCards** method takes as parameters the current player, a list of all players, the type of action card, and the list of Uno cards. It uses a switch statement to determine the type of action card and invokes the corresponding method to process the card's effect on the players.
2. Skip Card: If the action card type is SKIP, the **SkipCard.skipCard** method is called to skip the next player's turn and return the next player to continue the game.
3. Reverse Card: If the action card type is REVERSE, the **ReverseCard.reverseCard** method is called to reverse the order of play among the players and return the next player to continue the game.

4. Draw Two Card: If the action card type is DRAW_TWO, the **DrawTwoCard.drawTwoCard** method is called to make the next player draw two cards from the deck and return the next player to continue the game.
5. Handling Invalid Card Type: If the action card type is not one of the recognized types (SKIP, REVERSE, DRAW_TWO), an **IllegalArgumentException** is thrown, indicating an invalid card type.

12- DrawTwoCard:

```java
import java.util.List;

1 usage
public class DrawTwoCard {
    1 usage
    public static Player drawTwoCard(Player player, List<Player> players, List<UnoCard> unoCards) {

        Player nextP = PlayerInputHandler.getNextPlayer(player, players);
        System.out.println("Drawing two cards for " + nextP.getName());
        for (int i = 0; i < 2; i++) {
            UnoCard drawnCard = CardHandler.drawCardFromDeck(unoCards);
            nextP.addCardToHand(drawnCard);

        }
        System.out.println("Skipping the " + nextP.getName() + "'s turn.");

        Player p = PlayerInputHandler.getNextPlayer(nextP, players);
        return p;

    }
}
```

Description: The **DrawTwoCard** class is responsible for handling the effect of the Draw Two action card in the Uno game. It allows the player who played the Draw Two card to make the next player draw two cards from the deck and skip their turn.
Responsibilities:

1. Drawing Two Cards: The **drawTwoCard** method takes as parameters the current player, a list of all players, and the list of Uno cards. It retrieves the next player using the **PlayerInputHandler.getNextPlayer** method and displays a message indicating that two cards are being drawn for that player. It then iterates twice and adds two cards drawn from the deck using the **CardHandler.drawCardFromDeck** method to the next player's hand.
2. Skipping the Next Player's Turn: After drawing the cards, the method displays a message indicating that the next player's turn is being skipped.
3. Returning the Next Player: Finally, the method retrieves the player following the next player using the **PlayerInputHandler.getNextPlayer** method and returns that player. This ensures that the game continues with the appropriate player after the skipped turn.

13- ReverseCard:

```
public class ReverseCard {
    1 usage
    public static Player reverseCard(List<Player> players, Player player) {

        System.out.println("Reversing the order of play.");
        if (players.size() == 2) {

            System.out.println("play again!");
            return player;

        }

        List<Player> reversedPlayers = new ArrayList<>();
        for (int i = players.size() - 1; i >= 0; i--) {
            Player p = players.get(i);
            reversedPlayers.add(p);
        }

        System.out.println("the order before reversed: ");
        int w = 1;
        for (Player x : PlayerInputHandler.getPlayerOrder(players))
            System.out.println((w++) + " :" + x.getName());
        System.out.println();
        System.out.println("Order reversed!");
        System.out.println("the order will be :");
        int n = 1;
        for (Player y : reversedPlayers) {
            System.out.println((n++) + " :" + y.getName());
        }
        Player p = PlayerInputHandler.getNextPlayer(player, players);
        return p;

    }
}
```

Description: The **ReverseCard** class handles the effect of the Reverse action card in the Uno game. When a Reverse card is played, the order of play is reversed, meaning the direction in which players take turns is changed.
Responsibilities:

1. Reversing the Order of Play: The **reverseCard** method takes as parameters the list of players and the current player. It first checks if there are only two players in the game. In that case, it simply returns the current player, indicating that the turn order remains the same.
2. Reversing the Player Order: If there are more than two players, the method creates a new list called **reversedPlayers** and iterates over the original list of players in reverse order. It adds each player to the **reversedPlayers** list.
3. Displaying the Order Before Reversal: The method displays the original order of players before the reversal. It uses the **PlayerInputHandler.getPlayerOrder** method to retrieve the player order and prints it.
4. Displaying the Reversed Order: After reversing the player order, the method displays the new order of players. It iterates over the **reversedPlayers** list and prints each player's name along with their position in the order.
5. Determining the Next Player: Finally, the method retrieves the player following the current player using the **PlayerInputHandler.getNextPlayer** method and returns that player. This ensures that the game continues with the appropriate player after reversing the order of play.

14- SkipCard:

```
import java.util.List;

1 usage
public class SkipCard {
    1 usage
    public static Player skipCard(Player player, List<Player> players) {

        Player nextPlayer = PlayerInputHandler.getNextPlayer(player, players);
        System.out.println("Skipping the " + nextPlayer.getName() + "'s turn.");
        Player next = PlayerInputHandler.getNextPlayer(nextPlayer, players);

        return next;
    }
}
```

Description: The **SkipCard** class handles the effect of the Skip action card in the Uno game. When a Skip card is played, the next player's turn is skipped, and the turn passes to the player after them.
Responsibilities:

1. Skipping the Next Player's Turn: The **skipCard** method takes as parameters the current player and the list of players. It retrieves the next player using the **PlayerInputHandler.getNextPlayer** method and stores it in the **nextPlayer** variable. It then prints a message indicating that the next player's turn is being skipped.

2. Determining the Next Player after Skipping: The method retrieves the player following the next player using the **PlayerInputHandler.getNextPlayer** method and stores it in the **next** variable. This ensures that the game continues with the appropriate player after skipping a turn.
3. Returning the Next Player: The method returns the **next** player, indicating that it is their turn to play.

Object-Oriented Design

Let's examine how object-oriented design is implemented in my code

1. **Class Abstraction**: The code defines several classes, each representing a specific concept or entity in the game. For example, the **UnoCard**, **Player**, **Game**, **UnoDeck**, and **CardHandler** classes encapsulate related properties and behaviors.
2. **Encapsulation**: Encapsulation is employed to encapsulate the internal state of objects and provide controlled access to their data. Private access modifiers are used to hide the internal implementation details of the classes. Public methods (getters and setters) are provided to access and modify the data in a controlled manner.
3. **Inheritance**: The code utilizes inheritance to establish relationships between classes. For instance, the **UnoGameVariation** class inherits from the **Game** class, indicating that it is a specific type of game. Similarly, the **SkipCard**, **ReverseCard**, and **DrawTwoCard** classes extend the **Player** class, representing specialized types of cards with additional functionality.
4. **Polymorphism**: Polymorphism is demonstrated through polymorphic method calls and class hierarchy. The **CardHandler** class has methods that accept objects of the **UnoCard** class and perform operations based on their specific type. The **Game** class defines abstract methods that are implemented differently by subclasses, allowing for different behavior based on the specific game variation.
5. **Composition**: Composition is utilized to create complex objects by combining simpler objects. For example, the **Game** class has a list of **Player** objects, representing the composition of players in the game. The **UnoDeck** class contains a list of **UnoCard** objects, representing the composition of the deck.
6. **Polymorphic Method Calls**: Polymorphic method calls are made in the code, where methods like **processActionCard**, **processNumberCard**, and **processWildCard** accept **UnoCard** objects and invoke different behaviors based on the specific card type. This demonstrates the ability to treat objects of different classes as objects of a common superclass and dynamically invoke the appropriate method.
   By adhering to these object-oriented design principles, the code promotes modularity, reusability, and maintainability. It allows for the creation of well-organized and encapsulated objects, facilitating code understanding, extensibility, and ease of maintenance.

Design Patterns in Uno Card Game Implementation

In my implementation of the Uno card game, i have incorporated several design patterns to enhance the structure, flexibility, and extensibility of my codebase. The following design patterns have been employed:

1. Singleton Pattern
The Singleton pattern was utilized in the **PlayerInputHandler** class, which provides a central point of access for handling player input throughout the game. By enforcing that only one instance of **PlayerInputHandler** exists, we ensured consistent and synchronized handling of player actions, such as selecting cards or making game decisions.

2. Factory Method Pattern
The Factory Method pattern was employed in the **CardFactory** class, responsible for creating action card objects (Skip, Reverse, Draw Two) based on the provided **CardType**. The **actionCards** method acts as a factory method, encapsulating the creation logic and enabling the flexible generation of different types of action cards. This pattern promotes code modularity and extensibility by allowing easy addition of new action card types in the future.

3. Template Method Pattern
The Template Method pattern was utilized in the **Game** abstract class to define the overall structure of the Uno game. The abstract methods **initializeGame**, **isGameEnded**, **processTurn**, **processActionCard**, **processNumberCard**, and **processWildCard** serve as hooks that subclasses (specific Uno game variations) must implement. The **play** method acts as the template method, orchestrating the game flow by invoking the defined abstract methods. This pattern enables code reuse and provides a consistent framework for implementing different game variations.

4. Observer Pattern
The Observer pattern was employed in the interaction between the **Game** class and the **Player** class. The **Player** objects serve as observers, while the **Game** class acts as the subject. The subject notifies observers (players) of relevant game events, such as turn changes or card plays. This pattern ensures loose coupling between the game logic and player actions, allowing for easy addition of new players and providing a scalable architecture.

5. Iterator Pattern
The Iterator pattern was utilized in the **UnoDeck** class to iterate over the collection of Uno cards. By implementing a custom iterator, we enabled sequential access to the cards in a controlled manner. This pattern provides a uniform interface for accessing elements in a collection and allows for easy traversal without exposing the underlying structure of the deck.

By incorporating these design patterns into our Uno card game implementation, we achieved a well-structured, modular, and extensible codebase. Each pattern served a specific purpose, such as enforcing single instances, facilitating object creation, defining a game template, enabling communication between game components, and supporting efficient iteration over collections. Overall, the application of these design patterns contributed to the overall design quality and maintainability of our Uno card game implementation.

Defending the Codes against Principles

Let's examine how the codes align with these principles:

1. Single Responsibility Principle (SRP)
The codes exhibit a strong adherence to the SRP by assigning specific responsibilities to individual classes. Each class has a clear and well-defined purpose, such as managing players, handling cards, controlling game flow, or implementing game variations. This separation of concerns enhances code maintainability, as each class focuses on a single responsibility and can be easily understood, tested, and modified without impacting other parts of the system.

2. Open/Closed Principle (OCP)
The codes adhere to the OCP by embracing extensibility through the use of abstract classes and interfaces. For instance, the **Game** class defines abstract methods that can be implemented by different game variations. This design allows for the addition of new game types without modifying the existing codebase, promoting code stability and reducing the risk of introducing bugs in previously working functionalities. Additionally, the **CardFactory** class leverages the Factory Method pattern to support the OCP, enabling the seamless incorporation of new action card types without requiring modifications to existing code.

3. Dependency Inversion Principle (DIP)
The codes exhibit a strong adherence to the DIP by promoting loose coupling and dependency injection. The classes depend on abstractions rather than concrete implementations, enabling flexibility and interchangeability of components. For example, the **CardHandler** class relies on abstractions like **UnoCard** and **CardColor**, allowing for easy integration with different types of card games. This approach facilitates code maintenance and promotes code reusability by decoupling dependencies and ensuring that high-level modules depend on abstractions rather than low-level details.

4. Liskov Substitution Principle (LSP): The code follows LSP by adhering to the principle of substitutability. For example, the **UnoGameVariation** class, which extends the abstract **Game** class, can be used interchangeably with its parent class. This ensures that instances of the derived class can be substituted for instances of the base class without altering the correctness or behavior of the program.

Defending the Codes against "Effective Java" Items (Jushua Bloch):

Let's examine how the code aligns with some key "Effective Java" items:

1. **Item 1: Consider static factory methods instead of constructors:** The code demonstrates the use of static factory methods in the **CardFactory** class. By employing the **actionCards** method, specific action card types such as Skip, Reverse, and Draw Two are instantiated based on the provided **CardType**. This approach enhances the code's flexibility and descriptive power in creating action cards

2. **Item 5: Avoid creating unnecessary objects:** The **CardHandler** class demonstrates an adherence to this item. In the **drawCardFromDeck** method, a random card is selected from the **unoCards** list, efficiently avoiding the creation of unnecessary objects. By reusing existing **UnoCard** instances and removing them from the list upon drawing, the code promotes efficient resource utilization.

The relationship between classes:

1. UnoGameVariation - [**Inheritance**]-> Game: UnoGameVariation extends the Game class, indicating an inheritance relationship.

2. Game - [**Composition**] -> UnoDeck: The Game class has a composition relationship with the UnoDeck class, as it contains a reference to an UnoDeck object.

3. Game - [**Association**] -> CardHandler: The Game class has an association relationship with the CardHandler class, indicating that it interacts with CardHandler objects.

4. Game - [**Association**] -> PlayerInputHandler: The Game class has an association relationship with the PlayerInputHandler class, indicating that it interacts with PlayerInputHandler objects.

5. Player - [**Aggregation**] -> List<UnoCard>: The Player class has an aggregation relationship with a List of UnoCard objects, indicating that a Player has a collection of UnoCards.

6. CardHandler - [**Dependency**] -> UnoCard: The CardHandler class has a dependency on the UnoCard class, as it utilizes UnoCard objects in its methods.

7. CardHandler - [**Dependency**] -> CardColor: The CardHandler class has a dependency on the CardColor enum, as it uses CardColor objects in its methods.

In conclusion, the implementation of the Uno card game in Java demonstrates a well-structured and modular design. The code follows object-oriented principles, such as encapsulation, inheritance, and polymorphism, to represent different game elements and their interactions.

Throughout the development process, several design patterns were utilized to address common design challenges. The Factory Method pattern was employed in the CardFactory class to dynamically create action cards based on their type. The Singleton pattern was implemented in the UnoGameVariation class to ensure that only one instance of the game exists at a time. These patterns enhance the code's flexibility, extensibility, and maintainability.

The code also adheres to Clean Code principles proposed by Uncle Bob, emphasizing readability, maintainability, and code organization. Meaningful variable and method names, proper indentation, and consistent coding style contribute to the overall clarity of the codebase.

Furthermore, the code aligns with the guidelines outlined in Effective Java by Joshua Bloch. It utilizes immutable classes, such as CardColor and CardType enums, to ensure their state remains consistent. Effective exception handling, input validation, and defensive copying techniques are implemented to improve code reliability and robustness.

The adherence to SOLID principles is evident in the codebase, promoting loose coupling, high cohesion, and easy scalability. The classes demonstrate appropriate levels of abstraction and encapsulation, allowing for future enhancements and modifications without impacting the overall system.