
UNO
UNO Game
Engine

Table of Contents

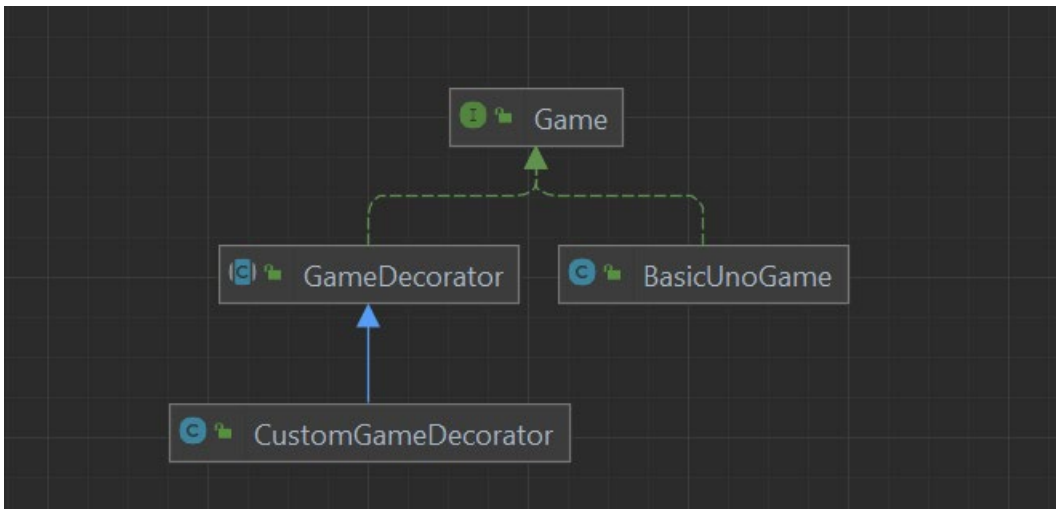
Introduction

This documentation provides a detailed overview of the UNO game implementation using the Decorator design pattern. UNO is a popular card game where players aim to get rid of their cards by matching them with the top card of the discard pile based on color or value.

Design Pattern

The Decorator design pattern is used in this implementation to add custom rules and properties to the basic UNO game. The Decorator pattern allows behavior to be added to an object dynamically at runtime by wrapping the object in a wrapper class. This enables the addition of new functionalities without modifying the existing code. The code utilizes the Decorator design pattern to extend the behavior of the basic UNO game. The Decorator pattern allows for adding custom rules and properties to the game without modifying the core implementation.

Class Diagram



Code Structure

The UNO game implementation consists of several classes:

Game (Interface)
play(): This method starts the game and drives the game flow.

```
public interface Game {
    3 usages  3 implementations
    void play();
}
```

BasicUnoGame (Class)

The BasicUnoGame class represents a basic implementation of the UNO game. It manages the game flow, including initializing the game, handling turns, and managing players. The class uses a deck of cards, a discard pile, and a list of players to simulate the game.

Class Members

- deck: Represents the deck of cards used in the game.
- discardPile: Represents the discard pile where played cards are placed.
- players: Represents a list of players participating in the game.
- currentPlayerIndex: Represents the index of the current player taking a turn.
- direction: Represents the direction of play (clockwise or counterclockwise).

Public Methods

- play(): Starts and runs the game until it ends. Manages turns, checks for winners, and handles player actions.
- initializeGame(): Initializes the game by getting the number of players, their names, shuffling the deck, dealing initial cards, and setting up the discard pile.
- dealCards(Player player, int numCards): Deals a specified number of cards from the deck to a player.
- getCurrentPlayer(): Returns the current player taking a turn.
- switchPlayer(): Switches the turn to the next player based on the direction of play.
- gameEnded(): Checks if the game has ended by determining if any player has an empty hand.
- playTurn(Player player): Manages the turn of a player, including displaying their hand, handling card selection, drawing cards, and updating the discard pile.
- handleSpecialCard(Card card): Handles the effects of special cards (Reverse, Skip, Draw Two, Wild, Wild Draw Four) played by a player.
- chooseColor(Player player): Allows a player to choose a color for a Wild card.
- getNextPlayer(): Returns the next player in line based on the direction of play.

```java
public void play() {
    initializeGame();

    while (!gameEnded()) {
        Player currentPlayer = getCurrentPlayer();
        playTurn(currentPlayer);

        if (currentPlayer.hasUno()) {
            System.out.println(currentPlayer.getName() + " says: Uno!");
        }

        if (currentPlayer.hasEmptyHand()) {
            System.out.println(currentPlayer.getName() + " wins!");
            break;
        }

        switchPlayer();
    }
}

1 usage
private void initializeGame() {
    Scanner scanner = new Scanner(System.in);

    // Handle number of players input exception
    int numPlayers = 0;
    boolean validNumPlayers = false;
    while (!validNumPlayers) {
        System.out.print("Enter the number of players (2-10): ");
        String input = scanner.nextLine();
```

```java
private void initializeGame() {
    Scanner scanner = new Scanner(System.in);

    // Handle number of players input exception
    int numPlayers = 0;
    boolean validNumPlayers = false;
    while (!validNumPlayers) {
        System.out.print("Enter the number of players (2-10): ");
        String input = scanner.nextLine();

        try {
            numPlayers = Integer.parseInt(input);
            if (numPlayers >= 2 && numPlayers <= 10) {
                validNumPlayers = true;
            } else {
                System.out.println("Invalid input! Please enter a number between 2 and 10.");
            }
        } catch (NumberFormatException e) {
            System.out.println("Invalid input! Please enter a valid number.");
        }
    }

    for (int i = 1; i <= numPlayers; i++) {
        // Handle player name input exception
        String name = "";
        boolean validName = false;
        while (!validName) {
            System.out.print("Enter the name of Player " + i + ": ");
            name = scanner.nextLine();

            if (name.matches( regex: "[a-zA-Z]+")) {
```

```
            if (name.matches( regex: "[a-zA-Z]+")) {
                validName = true;
            } else {
                System.out.println("Invalid input! Player name should only contain letters.");
            }
        }

        players.add(new Player(name));
    }

    deck.initialize();
    deck.shuffle();

    for (Player player : players) {
        dealCards(player, INITIAL_HAND_SIZE);
        //System.out.println(player.getName() + "'s hand: " + player.getHand());
    }


    Card topCard = deck.drawCard();
    while (topCard.getType() == Card.Type.WILD) {
        deck.addCardToBottom(topCard);
        topCard = deck.drawCard();
    }
    discardPile.addCard(topCard);
    System.out.println("Top card of the discard pile: " + topCard);
}
```

```
private void dealCards(Player player, int numCards) {
    for (int i = 0; i < numCards; i++) {
        try {
            Card card = deck.drawCard();
            player.drawCard(card);
        } catch (IllegalStateException e) {
            System.out.println("No cards are available. The deck is empty.");
            break; // Exit the loop if the deck is empty
        }
    }
}


3 usages
private Player getCurrentPlayer() { return players.get(currentPlayerIndex); }

5 usages
private void switchPlayer() {
    if (direction) {
        currentPlayerIndex = (currentPlayerIndex + 1) % players.size();
    } else {
        currentPlayerIndex = (currentPlayerIndex - 1 + players.size()) % players.size();
    }
}

1 usage
private boolean gameEnded() {
    for (Player player : players) {
        if (player.hasEmptyHand()) {
            return true;
        }
```

```
private boolean gameEnded() {
    for (Player player : players) {
        if (player.hasEmptyHand()) {
            return true;
        }
    }
    return false;
}

1 usage
public void playTurn(Player player) {
    System.out.println("--------");
    System.out.println("It's " + player.getName() + "'s turn");
    System.out.println(player.getName() + "'s hand: " + player.getHand()); // Print player's hand

    Scanner scanner = new Scanner(System.in);
    boolean drawInputValid = false;
    while (!drawInputValid) {
        System.out.print("Choose a card to play (or enter 'draw' to draw a card): ");
        String input = scanner.nextLine();

        if (input.equalsIgnoreCase( anotherString: "draw")) {
            Card drawnCard = deck.drawCard();
            player.drawCard(drawnCard);
            System.out.println("You drew a card: " + drawnCard);
            drawInputValid = true;
        } else {
            try {
                int index = Integer.parseInt(input);
                if (index >= 0 && index < player.getHandSize()) {
                    Card selectedCard = player.getCard(index);
```

```
        if (input.equalsIgnoreCase( anotherString: "draw")) {
            Card drawnCard = deck.drawCard();
            player.drawCard(drawnCard);
            System.out.println("You drew a card: " + drawnCard);
            drawInputValid = true;
        } else {
            try {
                int index = Integer.parseInt(input);
                if (index >= 0 && index < player.getHandSize()) {
                    Card selectedCard = player.getCard(index);
                    if (selectedCard.isValidMove(discardPile.getTopCard())) {
                        player.playCard(selectedCard);
                        discardPile.addCard(selectedCard);
                        System.out.println(player.getName() + " played: " + selectedCard);
                        handleSpecialCard(selectedCard);
                        drawInputValid = true;
                    } else {
                        System.out.println("Invalid move! Please try again.");
                    }
                } else {
                    System.out.println("Invalid input! Please try again.");
                }
            } catch (NumberFormatException e) {
                if (!input.equalsIgnoreCase( anotherString: "draw")) {
                    System.out.println("Invalid input! Please enter a valid card index or 'draw'.");
                } else {
                    System.out.println("Invalid input! Please enter 'draw' in the correct format.");
                }
            }
        }
    }
```

```
    System.out.println(player.getName() + "'s hand: " + player.getHand()); // Print player's hand
    System.out.println("Top card of the discard pile: " + discardPile.getTopCard());
    System.out.println("--------");
}


1 usage
private void handleSpecialCard(Card card) {
    switch (card.getType()) {
        case REVERSE:
            direction = !direction;
            System.out.println("Direction reversed!");
            switchPlayer();
            break;
        case SKIP:
            System.out.println("Turn skipped!");
            switchPlayer();
            break;
        case DRAW_TWO:
            Player nextPlayer = getNextPlayer();
            dealCards(nextPlayer, numCards: 2);
            System.out.println(nextPlayer.getName() + " drew 2 cards and their turn is skipped!");
            switchPlayer();
            break;
        case WILD:
            Player currentPlayer = getCurrentPlayer();
            Color chosenColor = chooseColor(currentPlayer);
            card.setColor(chosenColor);
            System.out.println(currentPlayer.getName() + " chose the color: " + chosenColor);
            Player nextPlayer = getNextPlayer();
            dealCards(nextPlayer, numCards: 2);
            System.out.println(nextPlayer.getName() + " drew 2 cards and their turn is skipped!");
            switchPlayer();
            break;
        case WILD:
            Player currentPlayer = getCurrentPlayer();
            Color chosenColor = chooseColor(currentPlayer);
            card.setColor(chosenColor);
            System.out.println(currentPlayer.getName() + " chose the color: " + chosenColor);
            break;
        case WILD_DRAW_FOUR:
            Player currentPlayer2 = getCurrentPlayer();
            Color chosenColor2 = chooseColor(currentPlayer2);
            card.setColor(chosenColor2);
            System.out.println(currentPlayer2.getName() + " chose the color: " + chosenColor2);
            Player nextPlayer2 = getNextPlayer();
            dealCards(nextPlayer2, numCards: 4);
            System.out.println(nextPlayer2.getName() + " drew 4 cards and their turn is skipped!");
            switchPlayer();
            break;
    }
}

2 usages
private Color chooseColor(Player player) {
    Scanner scanner = new Scanner(System.in);
    System.out.print(player.getName() + ", choose a color (RED, GREEN, BLUE, YELLOW): ");
    String input = scanner.nextLine().toUpperCase();
    Color chosenColor;
    switch (input) {
```

```java
2 usages
private Color chooseColor(Player player) {
    Scanner scanner = new Scanner(System.in);
    System.out.print(player.getName() + ", choose a color (RED, GREEN, BLUE, YELLOW): ");
    String input = scanner.nextLine().toUpperCase();
    Color chosenColor;
    switch (input) {
        case "RED":
            chosenColor = Color.RED;
            break;
        case "GREEN":
            chosenColor = Color.GREEN;
            break;
        case "BLUE":
            chosenColor = Color.BLUE;
            break;
        case "YELLOW":
            chosenColor = Color.YELLOW;
            break;
        default:
            System.out.println("Invalid color choice. Choosing RED by default.");
            chosenColor = Color.RED;
            break;
    }
    return chosenColor;
}
```

```java
2 usages
private Player getNextPlayer() {
    int nextPlayerIndex;
    if (direction) {
        nextPlayerIndex = (currentPlayerIndex + 1) % players.size();
    } else {
        nextPlayerIndex = (currentPlayerIndex - 1 + players.size()) % players.size();
    }
    return players.get(nextPlayerIndex);
}
}
```

GameDecorator (Abstract Class)

Extends the Game interface.
Acts as the base decorator for adding custom rules and properties to the basic UNO game.
Holds a reference to the decorated game.
Implements the play() method by delegating the call to the decorated game.

```java
public abstract class GameDecorator implements Game {
    2 usages
    private Game decoratedGame;

    1 usage
    public GameDecorator(Game decoratedGame) {
        this.decoratedGame = decoratedGame;
    }

    3 usages  1 override
    public void play() {
        decoratedGame.play();
    }
}
```

CustomGameDecorator (Class)

Extends the GameDecorator class.
Implements additional custom rules and logic for the UNO game.
Overrides the play() method to modify the behavior of the game.

```java
public class CustomGameDecorator extends GameDecorator {
    1 usage
    public CustomGameDecorator(Game decoratedGame) {
        super(decoratedGame);
    }

    // Implement additional custom rules and properties
    3 usages
    public void play() {
        // Custom game logic
        super.play(); // Invoke the basic game play method
    }
}
```

Card (Class)

Represents an UNO card with a color and value.
Provides methods to get and set the color, check if it is a valid move, and determine the type of the card.

```java
public class Card {
    8 usages
    public enum Type { NUMBER, SKIP, REVERSE, DRAW_TWO, WILD, WILD_DRAW_FOUR }

    7 usages
    private Color color;
    6 usages
    private Value value;

    10 usages
    public Card(Color color, Value value) {
        this.color = color;
        this.value = value;
    }

    1 usage
    public Color getColor() { return color; }

    1 usage
    public Value getValue() { return value; }

    2 usages
    public void setColor(Color color) { this.color = color; }


    1 usage
    public boolean isValidMove(Card topCard) {
        if (color == Color.BLACK || topCard.getColor() == color || topCard.getValue() == value) {
            return true;
        }
```

```
1 usage
public boolean isValidMove(Card topCard) {
    if (color == Color.BLACK || topCard.getColor() == color || topCard.getValue() == value) {
        return true;
    }
    return false;
}

2 usages
public Type getType() {
    if (color == Color.BLACK) {
        if (value == Value.WILD_DRAW_FOUR) {
            return Type.WILD_DRAW_FOUR;
        } else {
            return Type.WILD;
        }
    }
    switch (value) {
        case SKIP:
            return Type.SKIP;
        case REVERSE:
            return Type.REVERSE;
        case DRAW_TWO:
            return Type.DRAW_TWO;
        default:
            return Type.NUMBER;
    }
}

@Override
public String toString() { return color + " " + value; }
```

Color (Enum)

Defines the colors available for UNO cards.

```
public enum Color {
    2 usages
    RED, GREEN, BLUE, YELLOW, BLACK
}
```

Deck (Class)

Represents the deck of UNO cards.
Provides methods to initialize the deck, shuffle the cards, draw a card, and add a card to the bottom of the deck.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;

2 usages
public class Deck {
    23 usages
    private List<Card> cards;

    1 usage
    public Deck() { this.cards = new ArrayList<>(); }

    1 usage
    public void initialize() {
        cards.clear();

        // Add number cards
        for (Color color : Color.values()) {
            if (color != Color.BLACK) {
                for (int i = 0; i <= 9; i++) {
                    cards.add(new Card(color, Value.values()[i]));
                    cards.add(new Card(color, Value.values()[i]));
                }
            }
        }

        // Add action cards
        for (Color color : Color.values()) {
            if (color != Color.BLACK) {
                cards.add(new Card(color, Value.SKIP));
```

```java
        // Add action cards
        for (Color color : Color.values()) {
            if (color != Color.BLACK) {
                cards.add(new Card(color, Value.SKIP));
                cards.add(new Card(color, Value.SKIP));
                cards.add(new Card(color, Value.REVERSE));
                cards.add(new Card(color, Value.REVERSE));
                cards.add(new Card(color, Value.DRAW_TWO));
                cards.add(new Card(color, Value.DRAW_TWO));
            }
        }

        // Add Wild and Wild Draw Four cards
        for (int i = 0; i < 4; i++) {
            cards.add(new Card(Color.BLACK, Value.WILD));
            cards.add(new Card(Color.BLACK, Value.WILD_DRAW_FOUR));
        }
    }

    1 usage
    public void shuffle() {
        Random random = new Random();
        for (int i = 0; i < cards.size(); i++) {
            int j = random.nextInt(cards.size());
            Card temp = cards.get(i);
            cards.set(i, cards.get(j));
            cards.set(j, temp);
        }
    }
```

```java
4 usages
public Card drawCard() {
    if (cards.isEmpty()) {
        throw new IllegalStateException("No cards are available. The deck is empty.");
    }
    return cards.remove( index: cards.size() - 1);
}


1 usage
public void addCardToBottom(Card card) {
    cards.add( index: 0, card);
}

public int getNumRemainingCards() {
    return cards.size();
}
```

DiscardPile (Class)

Represents the discard pile of UNO cards.
Maintains a list of cards and provides methods to get the top card and add a card to the pile.

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

2 usages
public class DiscardPile {
    5 usages
    private List<Card> cards;

    1 usage
    public DiscardPile() {
        this.cards = new ArrayList<>();
    }

    2 usages
    public Card getTopCard() {
        if (cards.isEmpty()) {
            return null;
        }
        return cards.get(cards.size() - 1);
    }

    2 usages
    public void addCard(Card card) {
        cards.add(card);
    }
}
```

Player (Class)

Represents a player in the UNO game.
Has a name and a hand of cards.
Provides methods to get the player's name, hand, and hand size, draw and play cards, check for UNO, and determine if the hand is empty.

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

14 usages
public class Player {
    2 usages
    private String name;
    8 usages
    private List<Card> hand;

    1 usage
    public Player(String name) {
        this.name = name;
        this.hand = new ArrayList<>();
    }

    11 usages
    public String getName() {
        return name;
    }

    2 usages
    public List<Card> getHand() {
        return hand;
    }

    1 usage
    public int getHandSize() {
        return hand.size();
    }

    1 usage
    public Card getCard(int index) {
        return hand.get(index);
    }

    2 usages
    public void drawCard(Card card) {
        hand.add(card);
    }

    1 usage
    public void playCard(Card card) {
        hand.remove(card);
    }

    1 usage
    public boolean hasUno() {
        return hand.size() == 1;
    }

    2 usages
    public boolean hasEmptyHand() {
        return hand.isEmpty();
    }
}
```

Value (Enum

Defines the values available for UNO cards.

```
public enum Value {
    ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, SKIP, REVERSE, DRAW_TWO, WILD, WILD_DRAW_FOUR
}
```

GameDriver (Class)

Contains the main method to start the UNO game.
Instantiates the desired game object and calls the play() method.

```
public class GameDriver {
    public static void main(String[] args) {
        Game game = new CustomGameDecorator(new BasicUnoGame());
        game.play();
    }
}
```

Defending Against Clean Code Principles

The code demonstrates several practices aligned with clean code principles:

Readability: Meaningful variable and method names are used, improving code readability.
Single Responsibility Principle (SRP): Classes have clear responsibilities, such as Game for game management and Player for player-related operations.
Proper Indentation and Formatting: The code follows consistent indentation and formatting conventions, enhancing readability.
Exception Handling: The code includes exception handling for input validation and empty deck scenarios.

Defending Against "Effective Java" Items

The code aligns with some guidelines from "Effective Java" by Joshua Bloch:

Item 1: Static Factory Methods: The code doesn't extensively use static factory methods.
Item 2: Builder Pattern: The code doesn't require complex object creation or a builder pattern.
Item 3: Singleton Pattern: The code doesn't utilize the singleton pattern.
Item 4: Noninstantiability: Utility classes, such as Deck and DiscardPile, don't allow instantiation.
Item 8: Finalizers and Cleaners: The code doesn't rely on finalizers or cleaners for resource management.

Defending Against SOLID Principles

The code aligns with SOLID principles, which promote good software design and maintainability:

Single Responsibility Principle (SRP): Classes have focused responsibilities, such as Game, Deck, and Player.

Open/Closed Principle (OCP): The code is open to extension but closed to modification. The core implementation of the UNO game, represented by the BasicUnoGame class, remains unchanged while allowing for the addition of custom rules and properties through the use of game decorators.

Conclusion

The provided code demonstrates the implementation of an UNO game using the Decorator design pattern. By utilizing the Decorator pattern, the code allows for extending the core game functionality with custom rules and properties. The code follows object-oriented design principles, including encapsulation, inheritance, and polymorphism, and demonstrates practices aligned with clean code principles and some guidelines from "Effective Java." With further improvements and additions, the code can provide a more robust and customizable UNO game experience.