UNO Game Engine

By Faris Qassas

Contents

Introduction 4

Code and Design 5

- Cards 5
  - Card Class 5
  - Numbered Cards 5
  - Action cards 6
  - Wild Cards 6
  - Card enums 6
- Card Actions 8
- Factories 8
  - Registries 9
- Deck 10
  - Standard Deck 12
- Rules 12
  - Interfaces 13
  - Implementation 13
- Game Management 15
  - Sub managers 15
  - Game manager 16
- Player 18
- Game 18
  - Setup 19

Uno Variation Example 20

Game Driver 21

Adhering to Software Engineering Standards 22

- Design patterns 22
- SOLID Principles 23
- Applying Clean Code Principles (Uncle Bob) 23
- Following Effective Java Guidelines (Joshua Bloch) 24

Summary 24

Introduction

Ever heard of Uno? It's a card game, a really fun one, actually! The rules are simple and easy to learn, but what makes Uno even more exciting is how many variations exist. Different versions introduce new action cards, unique penalties, or even change how cards are dealt.

In this project, I'm not just coding Uno, I'm building an Uno Game Engine. This engine allows you to customize your own variation of Uno, set your own rules, play with a custom deck, and even introduce brand-new cards to the game. Instead of playing with the same rules every time, why not change things up and create something unique?

To achieve this, I designed the engine using Object-Oriented Programming principles, making it extensible and reusable. At its core is an abstract Game class, which provides a structured way to create new Uno variations with minimal effort. Developers can extend this class, implement their own rules, and define custom game mechanics.

This report explains how I built the Uno Game Engine, covering OOP design, design patterns, clean code principles, Effective Java best practices, and SOLID principles. The goal is to show how this engine is designed to be maintainable, scalable, and fun to experiment with.

Code and Design

We will start explaining each part of the code in detail and ill be providing parts of the code along the report.

Cards

Starting with the basic things we need for an Uno game, the Cards. In Uno, cards are the core elements of the game that players use to perform actions and interact with each other. To represent these cards programmatically, we start with an abstract Card class, which lays the foundation for different types of Uno cards.

Card Class

The Card class serves as the blueprint for all types of cards in the game. It holds the common properties shared by all cards, such as the color and type. Additionally, it allows cards to have specific actions (for special cards like Reverse, Skip, Wild).

The class structure is designed using the **Strategy Pattern** to manage dynamic behaviours for the cards, Actions are selected and executed at runtime enabling flexible gameplay changes without modifying the card structure. This makes it easy to introduce new actions or modify existing ones by simply adding new action types and behaviors.

```java
public abstract class Card {
    protected CardColor color;
    protected CardType type;
    protected Action action;

    public Card(CardColor color, CardType type, Action action) {
        this.color = color;
        this.type = type;
        this.action = action;
    }

    public CardColor getColor() {return color;}

    public void setColor(CardColor color) {this.color = color;}

    public CardType getType() {return type;}

    public Action getAction() {return this.action; }

    @Override
    public abstract String toString();
}
```

Numbered Cards

The Numbered cards are the most basic type of cards in Uno, representing the numbers from 0 to 9 in various colors. These cards don't have any special actions associated with them. They are created using the NumberedCard class, which stores the number and color. This class can easily be extended if needed to add more features (number ranges beyond 0-9 or extra properties for special numbered cards).

```java
public class NumberedCard extends Card {
    private int number;

    public NumberedCard(CardColor color, int number) {
        super(color, CardType.NUMBERED, null);
        this.number = number;
    }

    public int getNumber() {
        return number;
    }

    @Override
    public String toString() {
        return this.getColor() + " " + number;
    }
}
```

Action cards

Action cards introduce special behaviours that impact the gameplay, such as Reverse, Skip, and Draw Two. These cards are represented by the ActionCard class, which holds the action type (like Skip or Reverse) and an optional Action object that defines the specific behaviour.
One of the key advantages of the ActionCard class is that it can be extended easily. The ActionType enum already defines common actions like Reverse, Skip, and Draw Two, but this approach leaves room for adding new action types. If a new action type needs to be introduced in the game (for example, "Shuffle," "Change Direction"), it can be added to the ActionType enum without altering the existing structure.

```java
public class ActionCard extends Card {
    private final ActionType actionType;

    public ActionCard(CardColor color, ActionType actionType, Action action){
        super(color, CardType.ACTION, action);
        this.actionType = actionType;
    }

    public ActionType getActionType() {
        return actionType;
    }

    @Override
    public String toString() {
        return this.getColor() + " " + actionType;
    }
}
```

Wild Cards

Wild cards are the most powerful in Uno, allowing players to change the color or force opponents to draw extra cards. The WildCard class includes both Wild and Wild Draw Four types, each capable of holding an Action. Using the Strategy Pattern, actions are dynamically assigned, enabling flexible behavior. Like ActionCard, this class can be extended to introduce new wild card mechanics. By adding new WildType values, such as Wild Swap Hands, custom rules can be seamlessly integrated.

```java
public class WildCard extends Card {
    private final WildType wildType;

    public WildCard(WildType wildType, Action action) {
        super(CardColor.WILD, CardType.WILD, action);
        this.wildType = wildType;
    }

    public WildType getWildType() {
        return wildType;
    }

    @Override
    public String toString() {
        return this.getColor() + " " + wildType;
    }
}
```

Card enums

We use enums to define card properties and behaviors while following key software design principles. Encapsulation keeps related constants grouped, Open/Closed Principle allows easy extension without modifying existing code, and Type Safety ensures only valid values are used. This structure makes the game flexible and maintainable.

CardColor Enum

Defines the four primary colors for the cards and a special WILD color for Wild cards. This enum can be extended if needed to introduce new colors (e.g., "Purple" for a future version of the game).

```
public enum CardColor {
    RED,
    BLUE,
    GREEN,
    YELLOW,
    WILD;
}
```

CardType Enum

Defines the three types of cards: NUMBERED, ACTION, and WILD. If new types of cards are introduced in the future, such as SHUFFLE cards or CHALLENGE cards, they can be added to this enum.

```
public enum CardType {
    NUMBERED,
    ACTION,
    WILD;
}
```

ActionType Enum

Represents the types of actions that can be performed with Action cards, like Reverse, Skip, and Draw Two. This enum is crucial for flexibility, as new action types can be added easily by appending new values.

```
public enum ActionType {
    REVERSE("Reverse"),
    SKIP("Skip"),
    DRAWTWO("Draw Two");

    private final String displayName;

    ActionType(String displayName){
        this.displayName = displayName;
    }

    @Override
    public String toString() {
        return displayName;
    }
}
```

WildType Enum

Defines the types of Wild cards, such as Change Color and Draw Four. New wild card types can be easily added here to expand the game.

```
public enum ActionType {
    REVERSE("Reverse"),
    SKIP("Skip"),
    DRAWTWO("Draw Two");

    private final String displayName;

    ActionType(String displayName){
        this.displayName = displayName;
    }

    @Override
    public String toString() {
        return displayName;
    }
}
```

Card Actions

The card actions are implemented in separate classes, rather than being hardcoded into the Card class itself. This allows each card to have its corresponding behaviour assigned dynamically, promoting flexibility without the need to create new card types for every action. This structure leverages the **Strategy Pattern**, where different strategies (actions) are encapsulated in their own classes, and can be applied to cards as needed.

The Action interface defines a single method, apply(), to execute the action associated with a card. This ensures that the game remains extensible, adhering to the **Open-Closed Principle** by allowing new actions without altering existing code.

```java
public interface Action {
    void apply(GameManager gameManager);
}
```

For example, the SkipAction class implements the Action interface, defining the behavior for skipping the next player. When the apply() method is called, it triggers the skipNextPlayer() method in the TurnManager, encapsulating the action logic. This allows the game to remain open for new action types without altering existing code, adhering to the Open-Closed Principle and making the code more maintainable and extensible.

```java
package Actions;

import Actions.interfaces.Action;
import Setup.GameManager;

public class SkipAction implements Action {
    @Override
    public void apply(GameManager gameManager) {

 gameManager.getTurnManager().skipNextPlayer();}
}
```

Factories

In this design, we utilize the **Factory Method** design pattern, where we have different concrete factories such as NumberedCardFactory, ActionCardFactory, and WildCardFactory. Each of these factories is responsible for creating specific types of cards, allowing the system to handle different card creation scenarios without directly coupling them to the rest of the codebase.

The CardFactory interface defines a common createCard() method, which is implemented by each concrete factory. For example, WildCardFactory creates wild cards, ActionCardFactory creates action cards, and NumberedCardFactory creates numbered cards. This separation allows the card creation logic for each type to be isolated.

```java
public interface CardFactory {
    Card createCard();
}
```

The WildCardFactory is responsible for creating WildCard objects. It takes a WildType (like Wild or Wild Draw Four) and a WildCardRegistry, which helps retrieve the appropriate action for the wild card. When createCard() is called, it fetches the corresponding action from the registry and returns a new WildCard with the specified type and action. This approach ensures that wild cards are created in a structured and flexible way, allowing easy modifications and extensions.

```java
public class WildCardFactory implements CardFactory {
    private final WildType wildType;
    private final WildCardRegistry wildCardRegistry;

    public WildCardFactory(WildType wildType, WildCardRegistry wildCardRegistry) {
        this.wildType = wildType;
        this.wildCardRegistry = wildCardRegistry;
    }

    @Override
    public Card createCard() {
        Action cardAction = wildCardRegistry.getActionForWildType(wildType);
        return new WildCard(wildType, cardAction);
    }}
```

Registries

Why do we need registries for the cards? Without them, every time a new wild or action card is introduced, we'd have to create new factories or modify the existing ones, which quickly becomes messy and hard to maintain. The registries allow us to manage the card behaviors dynamically, so we can easily add new cards without touching the core logic. This keeps the game flexible, scalable, and much cleaner to work with as it grows. It's all about making future changes easier and keeping things organized.

The WildCardRegistry acts as a centralized store for wild card behaviors, mapping each WildType to its corresponding action using a Supplier<Action>. When a wild card is created, it looks up its action dynamically rather than relying on hardcoded logic. This keeps the system adaptable and scalable. Additionally, the registry includes error handling, if an unsupported WildType is requested, it throws an IllegalArgumentException. This prevents invalid wild card actions from being assigned and helps catch potential issues early.

```java
public class WildCardRegistry {
    private final Map<WildType, Supplier<Action>> wildCardActions = new HashMap<>();

    public void registerAction(WildType wildType, Supplier<Action> actionSupplier) {
        wildCardActions.put(wildType, actionSupplier);
    }

    public Action getActionForWildType(WildType wildType) {
        Supplier<Action> actionSupplier = wildCardActions.get(wildType);
        if (actionSupplier == null) {
            throw new IllegalArgumentException("Unsupported wild type: " + wildType);
        }
        return actionSupplier.get();
    }
}
```

The ActionRegistry follows a similar approach for action cards like SKIP, REVERSE, and DRAW_TWO. It registers actions dynamically and retrieves them when needed, keeping the factory logic clean and flexible. Error handling is also built in, if an unrecognized ActionType is requested, the registry throws an exception, preventing unexpected behavior. By enforcing strict type checking, both registries ensure that only valid actions are assigned, improving code reliability and maintainability.

```
public class ActionRegistry {
    private final Map<ActionType, Supplier<Action>> actionMap = new HashMap<>();

    public void registerAction(ActionType actionType, Supplier<Action> actionSupplier) {
        actionMap.put(actionType, actionSupplier);
    }

    public Action createAction(ActionType actionType) {
        Supplier<Action> actionSupplier = actionMap.get(actionType);
        if (actionSupplier == null) {
            throw new IllegalArgumentException("Unsupported action type: " + actionType);
        }
        return actionSupplier.get();
    }
}
```

To add new cards, you simply create the necessary action classes for them and register them in the respective registrie, either ActionRegistry or WildCardRegistry. The registries associate the new actions with their corresponding types, keeping the core logic untouched. This modular approach allows easy extensibility without modifying existing code. The registries handle the dynamic creation of cards, ensuring scalability and maintainability. By eliminating the need for creating new factories, we avoid the overhead of writing separate factories for each new card type. Instead, the existing factory classes can simply pull from the registries to inject the appropriate actions, streamlining the process.

Deck

The Deck is the collection of cards that players will use during the game, containing all the necessary cards to follow the rules of play. It provides essential functionality like shuffling, drawing, and managing the cards efficiently.

The DeckFactory interface defines a standard way to create decks, allowing different deck types (Uno, Poker) to be built consistently. It ensures flexibility and extension without altering core deck logic, with a single createDeck() method returning a Deck object.

```
public interface DeckFactory {
    Deck createDeck();
}
```

The Deck class represents a deck of cards and encapsulates all related operations. It maintains a list of Card objects, supporting essential functions such as adding, drawing, and shuffling cards. The shuffle() method performs multiple iterations of shuffling to ensure randomness, while drawCard() removes and returns the top card from the deck, throwing an exception if the deck is empty. The class also provides methods to check whether the deck is empty and retrieve all remaining cards in a safe, encapsulated manner. This separation of concerns ensures that the deck logic is modular and reusable.

```java
public class Deck {
    private List<Card> cards;

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

    public Deck(List<Card> initialCards) {
        this.cards = new ArrayList<>(initialCards);
    }

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

    public void addAllCards(List<Card> cardsToAdd) {
        cards.addAll(cardsToAdd);
    }

    public void shuffle() {
        for (int i = 0; i < 7; i++) {
            Collections.shuffle(cards);
        }
    }

    public Card drawCard() {
        if (cards.isEmpty()) {
            throw new IllegalStateException("The deck is empty!");
        }
        return cards.remove(0);
    }

    public List<Card> getCards() {
        return new ArrayList<>(cards);
    }

    public boolean isEmpty() {
        return cards.isEmpty();
    }
}
```

Standard Deck

The StandardUnoDeckFactory is responsible for constructing a complete Uno deck by following predefined rules. It utilizes the **Factory Pattern** to separate the deck creation logic from the Deck class, ensuring a clean and scalable design. This class depends on two registries ActionRegistry and WildCardRegistry to dynamically assign behavior to action and wild cards instead of hardcoding their logic. The createDeck() method initializes an Uno deck by calling helper methods: addNumberedCards(), which adds cards numbered 0-9 for each color; addActionCards(), which generates action cards like Skip, Reverse, and Draw Two; and addWildCards(), which adds special wild cards such as Wild and Wild Draw Four. By utilizing registries and factory methods, this approach allows easy modification and extension of card behavior without altering

the core deck creation logic.

```java
public class StandardUnoDeckFactory implements DeckFactory {
    private final ActionRegistry actionRegistry;
    private final WildCardRegistry wildCardRegistry;

    public StandardUnoDeckFactory(ActionRegistry actionRegistry, WildCardRegistry wildCardRegistry) {
        this.actionRegistry = actionRegistry;
        this.wildCardRegistry = wildCardRegistry;
    }

    @Override
    public Deck createDeck() {
        List<Card> cards = new ArrayList<>();
        addNumberedCards(cards);
        addActionCards(cards);
        addWildCards(cards);
        return new Deck(cards);
    }


    private void addNumberedCards(List<Card> cards) {
        for (CardColor color : CardColor.values()) {
            if (color != CardColor.WILD) {
                cards.add(new NumberedCardFactory(color, 0).createCard());
                for (int i = 1; i <= 9; i++) {
                    cards.add(new NumberedCardFactory(color, i).createCard());
                    cards.add(new NumberedCardFactory(color, i).createCard());
                }
            }
        }
    }

    private void addActionCards(List<Card> cards) {
        for (CardColor color : CardColor.values()) {
            if (color != CardColor.WILD) {
                for (ActionType action : ActionType.values()) {
                    cards.add(new ActionCardFactory(color, action, actionRegistry).createCard());
                    cards.add(new ActionCardFactory(color, action, actionRegistry).createCard());
                }
            }
        }
    }

    private void addWildCards(List<Card> cards) {
        for (WildType wildType : WildType.values()) {
            WildCardFactory wildCardFactory = new WildCardFactory(wildType, wildCardRegistry);
           for (int i = 0; i < 4; i++) {
                cards.add(wildCardFactory.createCard());
            }
        }
    }
}
```

Rules

The Rules system ensures that the game follows a structured set of guidelines, such as determining if a card can be played, applying penalties, and checking game-ending conditions. It is designed using interfaces to allow flexibility and easy expansion of game rules without modifying existing logic.

Interfaces

The Rules interface defines the core game logic, ensuring a card is playable, penalties are enforced, and the game ending condition is checked. By using this interface, different rule sets can be implemented for variations of the game while maintaining a unified structure.

```java
public interface Rules {
    boolean canPlayCard(Card cardToPlay, Card topCard);
    void applyPenalty(Player player, DeckManager deckManager);
    Player isGameOver(List<Player> players);
}
```

The Penalty interface abstracts the logic for applying penalties. This allows different types of penalties (drawing extra cards) to be implemented without modifying the core rule system.

```java
public interface Penalty {
    void apply(Player player, DeckManager deckManager);
}
```

This interface is responsible for determining the game's end. The determineWinner method checks the players' statuses to decide if one has met the winning conditions. By separating this logic, different game-ending rules can be implemented without affecting the main game structure.

```java
public interface GameOverCondition {
    Player determineWinner(List<Player> players);
}
```

The CardLegality interface ensures that the rules governing whether a card can be played are handled separately. The isLegal method determines if the current card can be played based on the top card in play. This separation allows flexibility in implementing different rule variations.

```java
public interface CardLegality {
    boolean isLegal(Card cardToPlay, Card topCard);
}
```

Implementation

Now that we've defined the basic interfaces, we can move on to implementing our own game rules. I'll be demonstrating one example of how we can apply the card legality rule in our game.

In this class we implemented the CardLegality interface, which defines the logic for determining whether a card is legal to play based on the top card of the deck. The StandardCardLegality class checks for different conditions: if the card is a WildCard (which can always be played), if the colors match, if both cards are ActionCards with the same action type, or if both cards are NumberedCards with the same number. If any of these conditions are met, the card is considered legal. Again, the rules are fully customizable and allows creativity from the developer perspective.

```java
public class StandardCardLegality implements CardLegality {
    @Override
    public boolean isLegal(Card card, Card topCard) {
        if (card instanceof WildCard) {
            return true;
        }

        if (card.getColor().equals(topCard.getColor())) {
            return true;
        }

        if (card instanceof ActionCard && topCard instanceof ActionCard) {
            ActionCard actionCard = (ActionCard) card;
            ActionCard actionTopCard = (ActionCard) topCard;

            if (actionCard.getActionType().equals(actionTopCard.getActionType())) {
                return true;
            }
        }

        if (card instanceof NumberedCard && topCard instanceof NumberedCard) {
            NumberedCard numberedCard = (NumberedCard) card;
            NumberedCard numberedTopCard = (NumberedCard) topCard;

            if (numberedCard.getNumber() == numberedTopCard.getNumber()) {
                return true;
            }
        }
        return false;
    }
}
```

Then, the StandardRules class combines multiple game rules. It adheres to the Rules interface and provides concrete implementations for the methods that check if a card can be played, apply penalties to a player, and determine if the game is over. By combining different rule interfaces, we can easily extend or modify the game rules without affecting the core game logic.
This approach provides flexibility, allowing the game rules to be managed in a modular way while adhering to the principles of the interface segregation and single responsibility patterns.

```java
public class StandardRules implements Rules {
    private final CardLegality cardLegality;
    private final Penalty penalty;
    private final GameOverCondition gameOverCondition;

    public StandardRules(CardLegality cardLegality,
                         Penalty penalty,
                         GameOverCondition gameOverCondition) {
        this.cardLegality = cardLegality;
        this.penalty = penalty;
        this.gameOverCondition = gameOverCondition;
    }

    @Override
    public boolean canPlayCard(Card card, Card topCard) {
        return cardLegality.isLegal(card, topCard);
    }

    @Override
    public void applyPenalty(Player player, DeckManager deckManager) {
        penalty.apply(player, deckManager);
    }

    @Override
    public Player isGameOver(List<Player> players) {
        return gameOverCondition.determineWinner(players);
    }
}
```

Game Management

Game management is all about bringing together all the classes we've implemented so far. It acts as the central controller, coordinating deck handling, turn progression, rule enforcement, and player actions. By managing these components seamlessly, it ensures smooth gameplay while maintaining flexibility for different rule sets and game variations.

In our implementation, we have both sub managers and a main game manager, each handling specific aspects of the game. To maintain flexibility, we utilize the **Strategy Pattern**, allowing different rules, deck management approaches, and turn-handling strategies to be easily swapped without modifying the core game logic. Let's go through these components.

Sub managers

The DeckManager interface defines essential deck operations like drawing a card, shuffling, and repopulating from the discard pile. It follows the **Strategy Pattern**, allowing different deck-handling strategies without modifying the game logic.

```
public interface DeckManager {
    Card drawCard();
    void shuffleDeck();
    void repopulateDeckFromDiscardPile(List<Card> discardPile);
}
```

The TurnManager interface manages turn flow with methods for advancing turns, skipping players, reversing order, and tracking the current player. By defining these in an interface, different turn management strategies can be implemented.

```
public interface TurnManager {
    void nextTurn();
    void skipNextPlayer();
    void reverseTurnOrder();
    int getCurrentPlayerIndex();
}
```

Let's look at an example implementation of the turn manager, which keeps track of players and manages turn progression. The nextTurn() method cycles through players, while skipNextPlayer() and reverseTurnOrder() adjust the turn sequence. This follows the **Iterator Pattern**, ensuring efficient player iteration.

```java
public class StandardTurnManager implements TurnManager {
    private final List<Player> players;
    private int currentPlayerIndex;
    private boolean isReversed = false;

    public StandardTurnManager(List<Player> players) {
        this.players = players;
        this.currentPlayerIndex = 0;
    }

    @Override
    public void nextTurn() {
        if (isReversed) {
            currentPlayerIndex = (currentPlayerIndex - 1 + players.size()) % players.size();
        } else {
            currentPlayerIndex = (currentPlayerIndex + 1) % players.size();
        }
    }

    @Override
    public void skipNextPlayer() {
        int nextPlayerIndex;
        if (isReversed) {
            nextPlayerIndex = (currentPlayerIndex - 1 + players.size()) % players.size();
        } else {
            nextPlayerIndex = (currentPlayerIndex + 1) % players.size();
        }
        String nextPlayerName = players.get(nextPlayerIndex).getName();
        System.out.println("Skipping " + nextPlayerName + "'s turn.");
        nextTurn();
    }

    @Override
    public void reverseTurnOrder() {
        System.out.println("Reversing the turn order.");
        isReversed = !isReversed;
    }

    @Override
    public int getCurrentPlayerIndex() {
        return currentPlayerIndex;
    }
}
```

Game manager

In any card game, managing the flow of the game is crucial. The *GameManager* acts as the central controller, ensuring that everything runs smoothly, from handling turns and enforcing rules to managing the deck and keeping track of player actions. Without a dedicated manager, the logic would be scattered across multiple classes, making the game harder to maintain and extend.

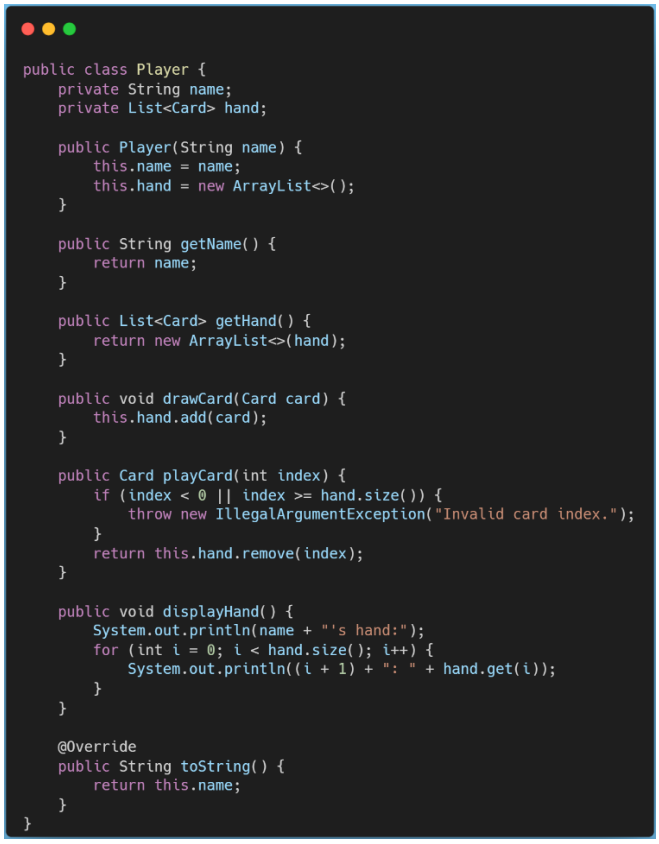
Our GameManager is implemented as a **Singleton**, meaning only one instance exists throughout the game. This ensures that all components interact with a single source of truth, preventing inconsistencies in game state. The getInstance() method enforces this by creating the instance only once and returning it whenever needed.
Instead of hardcoding specific game mechanics, we inject dependencies like DeckManager, TurnManager, and Rules. This allows different implementations of these components, making it easy to modify game rules or card behavior without changing the core logic.
The **Facade Pattern** plays a role here as well, GameManager provides a unified interface to interact with different parts of the game, simplifying how other classes access deck operations, turn progression, and rule enforcement. This keeps the codebase cleaner and easier to navigate.
Additionally, there's an implicit use of the **Observer Pattern** in the way actions affect the game state. When a card with a special action is played, its effect is triggered through an Action object, modifying turn order, skipping players, or enforcing penalties dynamically. This design makes it easier to extend the game with new actions in the future. Overall, GameManager brings everything together, ensuring smooth coordination between different components while keeping the design modular and adaptable.

```java
public class GameManager {
    private static GameManager instance;
    private final DeckManager deckManager;
    private final TurnManager turnManager;
    private final Rules rules;
    private final List<Player> players;
    private final List<Card> discardPile = new ArrayList<>();
    private final ColorSelector colorSelector;

    private GameManager(DeckManager deckManager, TurnManager turnManager, Rules rules, List<Player> players) {
        this.deckManager = deckManager;
        this.turnManager = turnManager;
        this.rules = rules;
        this.players = new ArrayList<>(players);
        this.colorSelector = new ColorSelector(new Scanner(System.in));
    }

    public static synchronized GameManager getInstance(DeckManager deckManager, TurnManager turnManager,
                                                       Rules rules, List<Player> players) {
        if (instance == null) {
            instance = new GameManager(deckManager, turnManager, rules, players);
        }
        return instance;
    }

    public void startGame() {
        System.out.println("Starting the game!");
        while (true) {
            playTurn();
            Player winner = rules.isGameOver(players);
            if (winner != null) {
                endGame(winner);
                break;
            }
            turnManager.nextTurn();
        }
    }

    public void playTurn() {
        Player currentPlayer = players.get(turnManager.getCurrentPlayerIndex());
        System.out.println("\nTop card: " + getTopCard() + " (Current color: " + getTopCard().getColor() + ")");
        System.out.println("It's " + currentPlayer.getName() + "'s turn.");
        currentPlayer.displayHand();

        int cardIndex = promptForCardIndex(currentPlayer);
        if (cardIndex == 0) {
            System.out.println(currentPlayer.getName() + " draws a card.");
            currentPlayer.drawCard(deckManager.drawCard());
            return;
        }

        Card chosenCard = currentPlayer.getHand().get(cardIndex - 1);
        if (!rules.canPlayCard(chosenCard, getTopCard())) {
            System.out.println("Illegal move!");
            rules.applyPenalty(currentPlayer, deckManager);
            return;
        }

        discardPile.add(chosenCard);
        currentPlayer.playCard(cardIndex - 1);

        Action action = chosenCard.getAction();
        if (action != null) {
            action.apply(this);
        }
    }


    public void endGame(Player winner) {
        System.out.println(winner.getName() + " has won the game!");
    }

    public Card getTopCard() {
        if (discardPile.isEmpty()) {
            throw new IllegalStateException("The discard pile is empty!");
        }
        return discardPile.get(discardPile.size() - 1);
    }

    public void setInitialTopCard(Card topCard) {
        discardPile.add(topCard);
    }

    public TurnManager getTurnManager() {
        return turnManager;
    }

    public DeckManager getDeckManager() {
        return deckManager;
    }

    public List<Player> getPlayers() {
        return new ArrayList<>(players);
    }

    public ColorSelector getColorSelector() {
        return colorSelector;
    }

    private int promptForCardIndex(Player currentPlayer) {
        System.out.println("Choose a card to play (enter the index) or enter 0 to draw a card:");
        while (true) {
            String input = System.console().readLine();
            try {
                int cardIndex = Integer.parseInt(input);
                if (cardIndex == 0) {
                    return 0;
                } else if (cardIndex > 0 && cardIndex <= currentPlayer.getHand().size()) {
                    return cardIndex;
                } else {
                    System.out.println("Invalid index. Please enter a valid card index or 0 to draw.");
                }
            } catch (NumberFormatException e) {
                System.out.println("Invalid input. Please enter a number.");
            }
        }
    }
}
```

Player

In any card game, players are the core participants, interacting with the deck and following the game's rules. The Player class represents an individual player, managing their hand of cards and providing essential actions like drawing and playing cards. Keeping this functionality encapsulated ensures that each player operates independently while still integrating seamlessly with the game logic.

Each player has a name and a hand, which is a list of Card objects. The constructor initializes these attributes, ensuring every player starts with an empty hand. The drawCard(Card card) method allows the player to receive a card from the deck, while playCard(int index) removes a selected card from their hand, enforcing basic validation to prevent invalid moves.
To enhance usability, the displayHand() method prints the player's current hand, making it easy to see available options.
By keeping the Player class simple and focused, we ensure flexibility in integrating different game mechanics while maintaining a clear and organized structure.



```java
public class Player {
    private String name;
    private List<Card> hand;

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

    public String getName() {
        return name;
    }

    public List<Card> getHand() {
        return new ArrayList<>(hand);
    }

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

    public Card playCard(int index) {
        if (index < 0 || index >= hand.size()) {
            throw new IllegalArgumentException("Invalid card index.");
        }
        return this.hand.remove(index);
    }

    public void displayHand() {
        System.out.println(name + "'s hand:");
        for (int i = 0; i < hand.size(); i++) {
            System.out.println((i + 1) + ": " + hand.get(i));
        }
    }

    @Override
    public String toString() {
        return this.name;
    }
}
```

Game

The Game class is the base for creating any Uno variation. It contains the essential components of a game, such as players, rules, and the deck. Using the **Template Method Pattern**, it provides a setup() method that allows subclasses to define specific initialization steps while keeping the core structure consistent and reusable.

```java
public abstract class Game {
    protected DeckManager deckManager;
    protected List<Player> players;
    protected Rules rules;

    public Game(List<Player> players, Rules rules, DeckManager
deckManager) {
        this.players = new ArrayList<>(players);
        this.rules = rules;
        this.deckManager = deckManager;
    }

    public void setup() {
        // No setup required by default
    }

    public DeckManager getDeckManager() {
        return deckManager;
    }

    public List<Player> getPlayers() {
        return new ArrayList<>(players);
    }
}
```

Setup

The GameSetupFactory class is responsible for setting up an Uno game by handling player input, initializing rules, and configuring the deck. Instead of creating separate factories for each card type, it uses **registries** (**ActionRegistry** and **WildCardRegistry**) to manage special card behaviors dynamically. This approach avoids the need for multiple card factories whenever a new action or wild card is introduced, making the system more scalable and maintainable. The registries map card types (Skip, Reverse, Draw Two, Wild) to their respective actions, allowing new mechanics to be added without modifying existing factories. A DeckFactory then generates a complete Uno deck using these registries. The game is set up with players drawing their starting hands, and a DeckManager and TurnManager are initialized to handle gameplay. Finally, a GameManager singleton is created, linking all components for a fully configured game environment.

```java
public class GameSetupFactory {
    public static GameManager createGameManager(Scanner scanner) {
        // Step 1: Get the number of players and their names
        System.out.println("Enter the number of players:");
        int numPlayers = scanner.nextInt();
        scanner.nextLine();
        List<Player> players = new ArrayList<>();
        for (int i = 1; i <= numPlayers; i++) {
            System.out.println("Enter the name of Player " + i + ":");
            String playerName = scanner.nextLine();
            players.add(new Player(playerName));
        }

        // Step 2: Initialize the rules
        Rules ruleSet = new StandardRules(
                new StandardCardLegality(),
                new StandardPenalty(),
                new StandardGameOverCondition()
        );

        // Step 3: Initialize the card registries
        ActionRegistry actionRegistry = new ActionRegistry();
        actionRegistry.registerAction(ActionType.SKIP, SkipAction::new);
        actionRegistry.registerAction(ActionType.REVERSE, ReverseAction::new);
        actionRegistry.registerAction(ActionType.DRAWTWO, DrawTwoAction::new);

        WildCardRegistry wildCardRegistry = new WildCardRegistry();
        wildCardRegistry.registerAction(WildType.WILD, WildChangeColorAction::new);
        wildCardRegistry.registerAction(WildType.WILD_DRAW_FOUR,
                                        WildDrawFourAction::new);

        // Step 5: Create the deck factory with both registries
        DeckFactory deckFactory = new StandardUnoDeckFactory(actionRegistry,
                                                             wildCardRegistry);

        // Step 6: Set up the game with the initial hand size
        int initialHandSize = 7;
        StandardUnoGame standardUnoGame = new StandardUnoGame(players, ruleSet,
                                                              deckFactory,
                                                              initialHandSize);
        standardUnoGame.setup();

        // Step 7: Initialize the managers
        DeckManager deckManager = new StandardDeckManager(standardUnoGame.getDeck());
        TurnManager turnManager = new StandardTurnManager(players);

        // Step 8: Create and return the GameManager
        GameManager gameManager = GameManager.getInstance(deckManager, turnManager,
                                                          ruleSet, players);
        gameManager.setInitialTopCard(standardUnoGame.getDiscardPile().get(0));
        return gameManager;
    }
}
```

Uno Variation Example

Now, coming to making our variation of Uno, the StandardUnoGame class is the core that sets up the entire game. It extends the abstract Game class, making sure everything from the deck creation to player hands is properly initialized. First, it sets up the deck using a DeckFactory, which ensures the deck can be easily customized if needed. Then, the game shuffles the deck and deals cards to each player, giving them their initial hand. One cool feature here is that if the first card drawn is a WildCard, the game reshuffles and draws again, ensuring we don't start the game with a special card. The discard pile tracks the played cards, and a StandardDeckManager handles all the deck operations like drawing, shuffling, and repopulating when necessary. This setup gives us the structure for a classic Uno game with room for future tweaks and changes.

```java
public class StandardUnoGame extends Game {
    private final int initialHandSize;
    private final List<Card> discardPile = new ArrayList<>();
    private final Deck deck;

    public StandardUnoGame(List<Player> players, Rules rules,
                           DeckFactory deckFactory, int initialHandSize)
{
        super(players, rules, null);
        this.initialHandSize = initialHandSize;
        this.deck = deckFactory.createDeck();
        this.deckManager = new StandardDeckManager(deck);
    }

    @Override
    public void setup() {
        getDeckManager().shuffleDeck();

        for (Player player : getPlayers()) {
            for (int i = 0; i < initialHandSize; i++) {
                player.drawCard(getDeckManager().drawCard());
            }
        }


        Card topCard = getDeckManager().drawCard();
        while (topCard instanceof WildCard) {
            getDeckManager().repopulateDeckFromDiscardPile(discardPile);
            getDeckManager().shuffleDeck();
            topCard = getDeckManager().drawCard();
        }
        discardPile.add(topCard);
    }

    public List<Card> getDiscardPile() {
        return new ArrayList<>(discardPile);
    }

    public Deck getDeck() {
        return deck;
    }
}
```

Game Driver

Ending with the GameDriver class, It serves as the entry point for running and testing our Uno game. With just two lines of code, it creates a Scanner for user input and utilizes the GameSetupFactory to set up the game. The GameManager is initialized, and the game starts by calling startGame(). It's a simple and clean way to kick off the entire game flow, making sure that all components, players, rules, deck, and managers are properly set up and ready to go. This is the file you'd run to see everything in action and test how our Uno game works.

```java
public class GameDriver {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        GameManager gameManager = GameSetupFactory.createGameManager(scanner);
        gameManager.startGame();
        scanner.close();
    }
}
```

Adhering to Software Engineering Standards

In this section, we will analyze the design choices made in our project, focusing on how we applied object-oriented design principles, utilized design patterns, and adhered to best practices from industry standards. We will also justify our implementation against Clean Code principles (Uncle Bob), Effective Java (Joshua Bloch), and SOLID principles to ensure maintainability, readability, and scalability.

Design patterns

Factory Pattern: Used in *GameSetupFactory*, *DeckFactory*, and *Card Factories* to abstract the creation of game components, ensuring a centralized, modular, and scalable approach while keeping the code flexible for future extensions.

Singleton Pattern: Applied in *GameManager* to ensure only one instance controls the game flow.

Strategy Pattern: Utilized in the action system for example, where different card actions (Skip, Draw Two) implement a common interface, enabling flexible and interchangeable behaviors without altering game logic.

Template Method Pattern: Implemented in *Game* to define a standard game structure while allowing variations through subclassing.

Facade Pattern: Implemented in *GameManager* to provide a unified interface for managing game components, simplifying interactions between different subsystems.

Registry Pattern: Used instead of multiple factories to manage card actions dynamically, allowing easy extension of new card types without modifying existing logic.

SOLID Principles

- **S (Single Responsibility Principle - SRP):** Each class has a single well-defined responsibility (DeckManager only manages deck operations).
- **O (Open/Closed Principle - OCP):** The Registry Pattern allows adding new actions without modifying existing logic, keeping code open for extension but closed for modification.
- **L (Liskov Substitution Principle - LSP):** Subclasses like StandardUnoGame properly extend Game, ensuring they can replace the parent class without breaking behavior.
- **I (Interface Segregation Principle - ISP):** Small, specific interfaces (DeckFactory, Rules, TurnManager) prevent unnecessary dependencies.
- **D (Dependency Inversion Principle - DIP):** High-level modules do not depend on low-level modules; instead, we rely on interfaces and abstract factories.

Applying Clean Code Principles (Uncle Bob)

- ✓ **Meaningful Names:** Classes, methods, and variables are given descriptive names that clearly convey their purpose (StandardUnoGame, GameSetupFactory), improving readability and reducing ambiguity.
- ✓ **Single Responsibility Principle (SRP):** Each class is designed with a specific, well-defined role, ensuring modularity and preventing bloated, multipurpose classes. For example, GameSetupFactory handles game setup, while GameManager orchestrates game flow.
- ✓ **Short and Focused Methods:** Methods are kept concise and adhere to the **single responsibility** principle, improving readability, maintainability, and ease of debugging.
- ✓ **Eliminating Code Duplication:** By leveraging **factories** and **registries**, we avoid redundant object creation logic, promoting reusability and reducing maintenance overhead.
- ✓ **Encapsulation and Data Hiding:** Internal implementation details are properly encapsulated within classes, ensuring that interactions occur through well-defined interfaces. This enhances modularity and prevents unintended modifications from external classes.

Following Effective Java Guidelines (Joshua Bloch)

- ✓ **Favoring Composition Over Inheritance:** Instead of deep inheritance hierarchies, we use composition, such as delegating deck operations to *DeckManager*.
- ✓ **Using Interfaces for Flexibility:** Interfaces like *DeckFactory*, Rules, and *TurnManager* allow easy swapping of implementations.
- ✓ **Minimizing Mutability:** Where possible, objects are made immutable (*discardPile* uses a copy to prevent unintended modifications).
- ✓ **Consistent Object Creation:** The Factory Pattern ensures controlled instantiation instead of relying on direct constructors.
- ✓ **Using final Where Applicable:** Helps maintain stability by preventing accidental modifications.

Summary

That was quite a journey. We really dived into how software is developed using Java—not just writing code, but writing clean, flexible, and easily maintained code. It took a lot of time, effort, and thought, but the final result is a structured and well-engineered project.

This report aimed to clearly explain the reason behind our design choices and how they contribute to a more robust system. Hopefully, it delivers a clear picture of how we approached building this game, not just from a functional standpoint but with a strong focus on long-term maintainability.