Atypon Training Program 2023

A WELL STRUCTURED ‘UNO GAME ENGINE’ ATYPON TRAINING PROGRAM 2023 USING OBJECT ORIENTED PROGRAMMING AND DESIGN PATTERNS

BY: YUSSIF ABDALLA, The University of Jordan

I. ABSTRACT

This report illustrates the solution of the Atypon’s 2023 assignment. The assignment is based on the famous UNO Card game. Uno is a card game that is typically played by few players, typically 2-10 players, and contains 108 cards divided into three types. Numbered cards (0-9), Action cards (Reverse, Skip, Draw Two) and Wild cards (Wild, Wild Draw Four).

An UNO game engine is successfully built using Java and OOP language. The program offers a flexibility to the developers to build their own variation of the UNO game. The programed code adhered to state of the art design techniques such as: Design patterns, Clean code, Effective Java, and SOLID principles. Finally, the programmed code is tested and verified successfully.

II. INTRODUCTION

A card game is any game using playing cards as the primary device with which the game is played, be they traditional or game-specific. The Uno game is considered a shedding game, where players start with a hand of cards, and the object of the game is to be the first player to discard all cards from one's hand [1].

A Java OOP Uno game engine is presented, which may be used by other developers (i.e., not players) and the players may build their own variation of an Uno game. The proposed solution creates an abstract class, which is called ‘Game’. Consequently, a class is used to create a new game variation, where a developer only needs to create a new class that extends the abstract Game class. The program is designed such that Developers can choose from predefined rules when creating their own game variation and they can extend the code and add more rules to it. The program also offers to the Developers ways to create new cards and new card-dealing mechanisms with minimal effort [2].

The program is engineered to have a class that is called GameDriver, which contains the main method. This main method contained two lines of code, one to instantiate a game object, and another one is used to invoke the play method.

This report presents a detailed step-by-step problem-solving approach, outlining the different classes created and the relationships among them. In addition, different design patterns are presented that are used and the process behind each decision made is highlighted. The report also summarizes the optimizations techniques that were implemented and the iterative evolution of the UNO game engine during the program's development.

III. OBJECT ORIENTED DESIGN

Object-Oriented Programming (OOP) is a suitable programming paradigm for developing the UNO game engine. The engine consists of many classes, each with its own specific attributes and methods that are dedicated to individual tasks. This approach aids in organizing and structuring the code, allowing for encapsulation, abstraction, reusability, inheritance, and polymorphism to be leveraged effectively.

First as a conceptual design is presented as a high level UML diagram, which is created to represent the classes of the solution and it is shown in Figure (1). This diagram defines the main classes that are used, their attributes, and the interactions between components, ensuring that the program meets the desired requirements and constraints.

3.1 The Uno Game Program Flow Chart

A flow chart of the proposed program is depicted in Figure (1). The program starts by creating an instant of a new game, then executes a method called game.play(). This play method starts by configuring the Uno game variations. Namely these variations include the following game setup methods, which may be overridden by developers.

addCards(): Add cards to the deck

assignCardToAction(): Executable cards are given actions

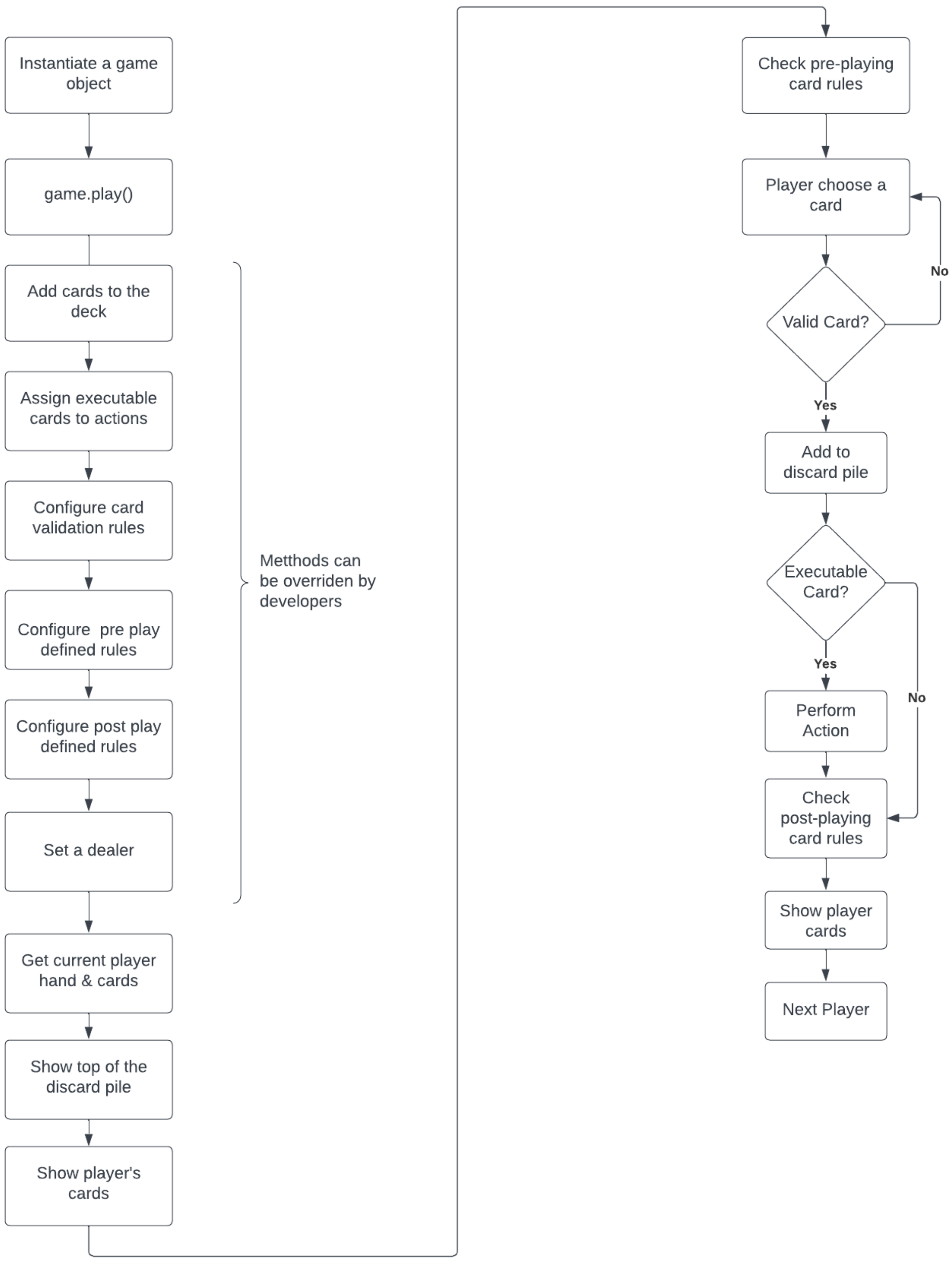
addValidationRules(): Validates cards against the discard pile

prePlayHandRules(): Configuring Pre rules

postPlayHandRules(): Configuring Post rules

setDealer(): Distribution of cards among players

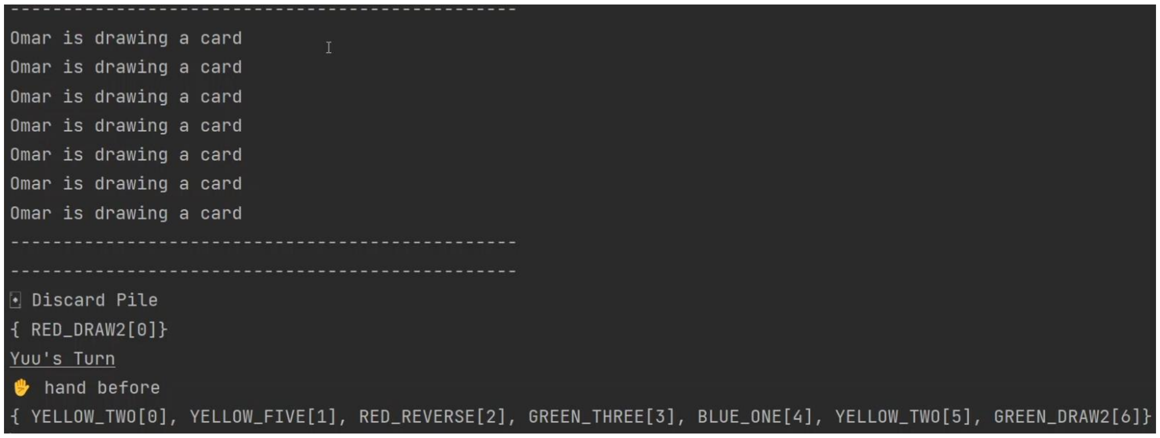
Once the game variation is set the Uno game may start by getting the current player's hand and cards. Then the player is shown the card at top of the pile. Referee and pre checks for the rules are performed before choosing a card. Then the user is allowed to select a card, if it is valid it will be accepted and executed. Furthermore, a post referee and checks for the rules are conducted, then the next player get the chance to select a card.



Instantiate a game object
game.play()
Add cards to the deck
Assign executable cards to actions
Configure card validation rules
Configure pre play defined rules
Configure post play defined rules
Set a dealer
Metthods can be overriden by developers
Get current player hand & cards
Show top of the discard pile
Show player's cards
Check pre-playing card rules
Player choose a card
Valid Card?
No
Yes
Add to discard pile
Executable Card?
Yes
No
Perform Action
Check post-playing card rules
Show player cards
Next Player

Figure 1: The Uno Game Flow Chart

Finally, this process iterates until one of the post rules that is designed to end the game applies and the game will terminate. Figure (2) shows a screen shot of the programed Uno game.



```
------------------------------------------------
Omar is drawing a card
Omar is drawing a card
Omar is drawing a card
Omar is drawing a card
Omar is drawing a card
Omar is drawing a card
Omar is drawing a card
------------------------------------------------
------------------------------------------------
🃏 Discard Pile
{ RED_DRAW2[0]}
Yuu's Turn
✋ hand before
{ YELLOW_TWO[0], YELLOW_FIVE[1], RED_REVERSE[2], GREEN_THREE[3], BLUE_ONE[4], YELLOW_TWO[5], GREEN_DRAW2[6]}
```

Figure 2: The Uno Game screen shot

3.2 High level UML diagram

Figure (3) depicts the program's high-level OOP design that was used in solving the given assignment. Referring to the high level UML diagram, the Game class is depicted as the central point among all classes and can be instantiated by the Game Driver class. <u>However, it's important to note that this diagram provides a high-level overview, illustrating the relationships between the Game class and other key classes, and it does not show the specifics of how other components interact with each other.</u>

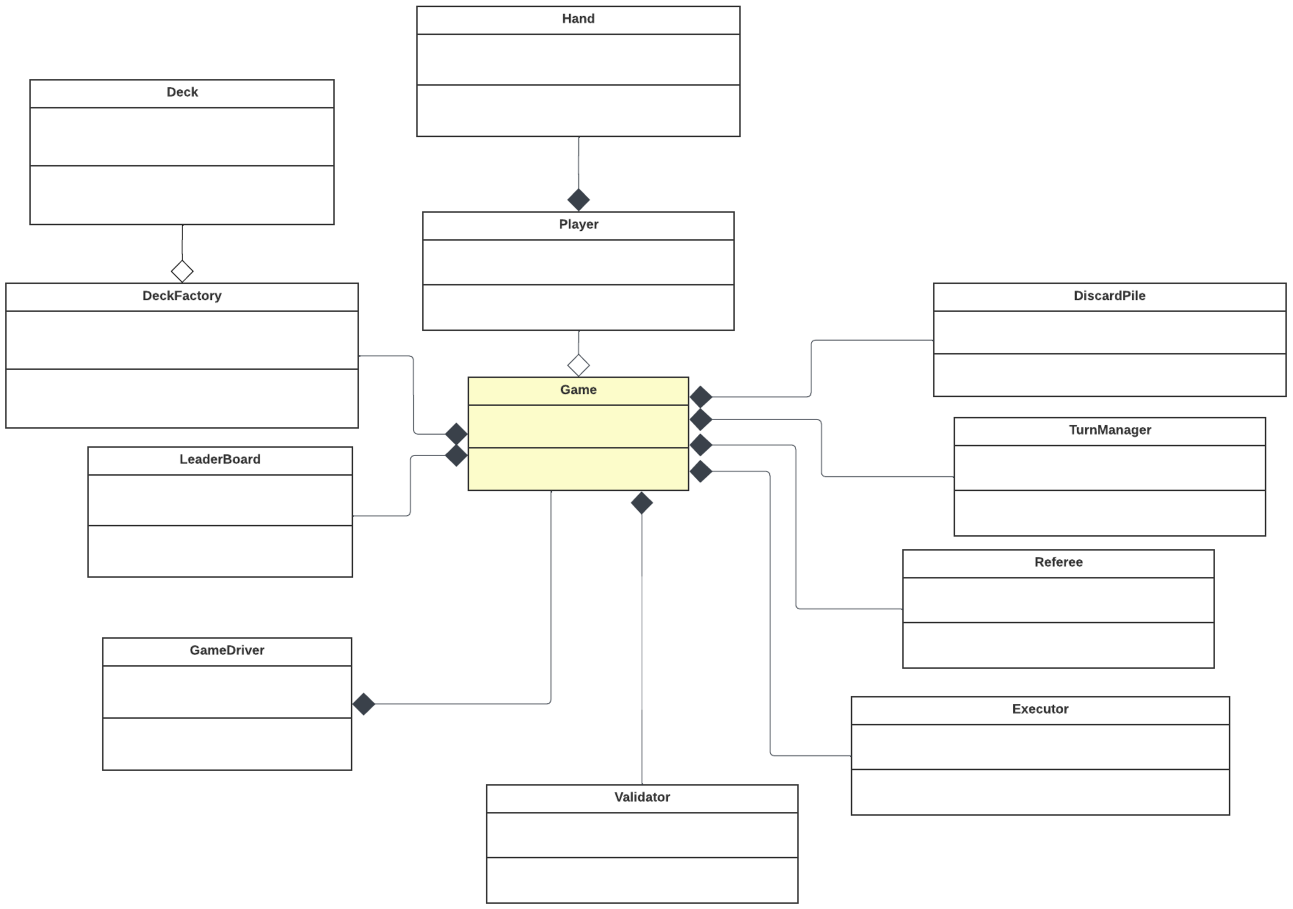


Figure 3: High-level UML Diagram of UNO Game Engine

3.3 Overview of the UNO Game main classes

In this section, a comprehensive list of the major used classes is overviewed. Consequently, a Game instance should include instances of these classes.

Player Class

According to the UNO game rules, the game can accommodate 2-10 players. Each player possesses a name, score, and a hand of cards.

Hand Class

A hand class represents the collection of cards held by a player throughout the game. Each player has their own hand, consisting of a deck from which they draw cards and a discard pile where they play the cards.

DiscardPile Class

A discard pile class represents the collection of cards played by players throughout the game. As players play cards, they are added to the top of the pile, with the most recently played card and they become the card at the top of the discard pile. Figure 2 depicts an illustration of this DiscardPile class.

Deck Class

The deck serves as a stack of cards from which players draw cards at the beginning of the game. Additionally, it acts as the source from which players draw cards when they receive a penalty.

TurnManager Class

The TurnManager class effectively handles the game's flow, managing player turns and smoothly adjusting the gameplay direction.

Referee Class

The Referee class ensures that players adhere to the Uno game rules throughout the game. It offers the flexibility to enforce pre-playing card rules, which monitor a player's hand before playing a card, as well as post-playing card rules, which monitor a player's hand after playing a card.

Validator Class

The Validator class examines the card selected by the player to determine its validity. It ensures that only valid cards are permitted, preventing any invalid cards to be played.

Executor Class

The Executor class applies the corresponding action of a card if the card possesses one. It activates when action or wild cards are encountered, allowing the execution of their specific effects. However, the Executor class does not interact with numbered cards, as they do not have associated actions.

DeckFactory Class

The DeckFactory class is responsible for generating new cards and supplying them to the deck. By default, it includes the standard UNO cards. However, developers have the flexibility to create new cards and add them to the Deck Factory.

LeaderBoard Class

The LeaderBoard class takes on the task of calculating and updating player's scores throughout the game. Upon reaching the end of the game, the leaderboard generates a table displaying the player's names alongside their respective scores.

Figure 4: The Deck and discard pile

3.4 UNO Cards

Card Class

The card is the base class for all types of cards in the game, which is show in Figure (5). It provides common properties and methods that are shared by all cards, such as a name and a value. Please note that all types of cards extend the Card class.

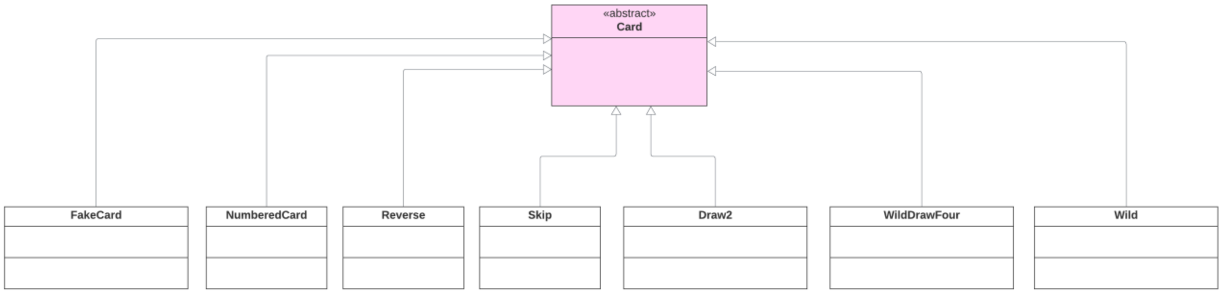


Figure 5: The Card class

The coloredCard Interface

This interface is implemented by cards that have a specific color, which is shown in Figure (6). It serves the purpose of defining a common behavior for cards that are associated with a color. The getColor() method declared in the ColoredCard interface allows retrieving the color of a card.

One of the main reasons for creating the ColoredCard interface is to facilitate the validation process performed by the Validator class. The Validator class is responsible for validating whether a card chosen by a player can be played on top of the discard pile. By utilizing the ColoredCard interface, the Validator can check if both the chosen card and the top card of the discard pile implement the ColoredCard interface and if they have the same color.

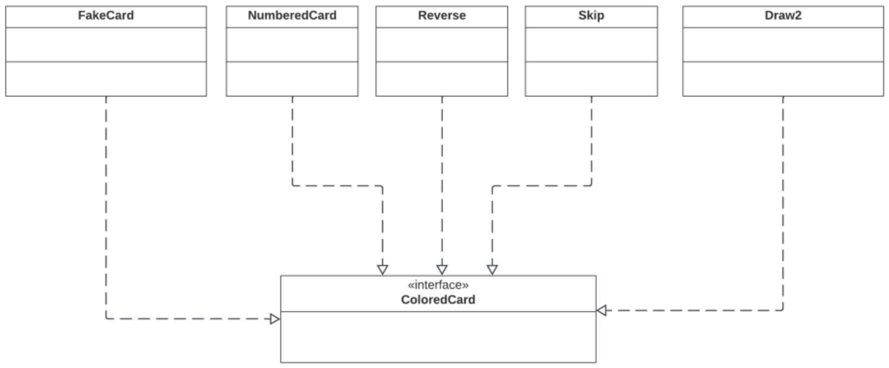


Figure 6: The Colored Card interface

The ExucatableCard Interface

The ExecutableCard interface is implemented by cards that have an associated action. It signifies that these cards can be executed to perform a specific action in the game. The purpose of this interface is to provide a common behavior for cards that have executable actions.

By implementing the ExecutableCard interface, a card indicates that it has a defined action that can be performed when played by the user. This allows the game executor to identify and handle these cards accordingly. The ExecutableCard interface typically includes a method, such as performAction(), that specifies the action to be executed when the card is played.

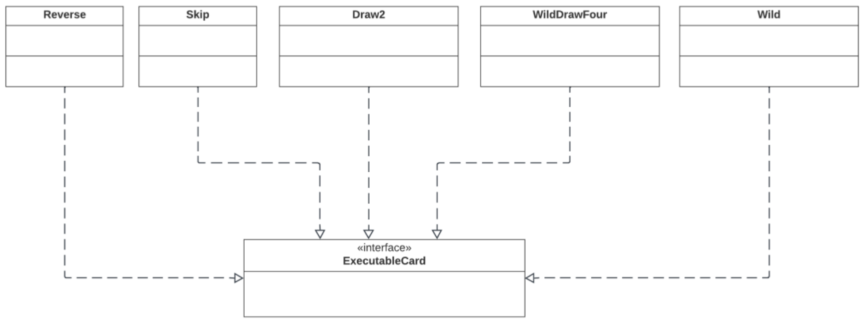


Figure 7: The Executable Card Interface

The WildCard Interface

This interface is implemented by cards that are considered wild cards, which is shown in Figure(8). Wild cards possess special properties or abilities that can have an impact on the game. The purpose of this interface is to identify and distinguish cards that are automatically accepted to play without the need to match anything with the discard pile.

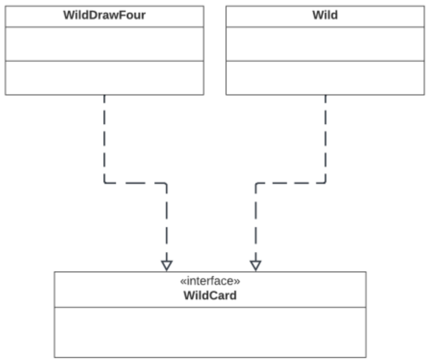


Figure 8: The Wild Card Interface

The Number and Color Enumerations

The ColoredCard interface has a method getColor(), which returns a value of the Color enum. The NumberedCard class, which likely implements ColoredCard, also has a method getNumber() to get the number of the card, which is represented by the Number enum. Figure (9) shows the number and color enumerations method.

Both Color and Number are independent enums with their respective values representing different colors and numbers that can be associated with the cards.

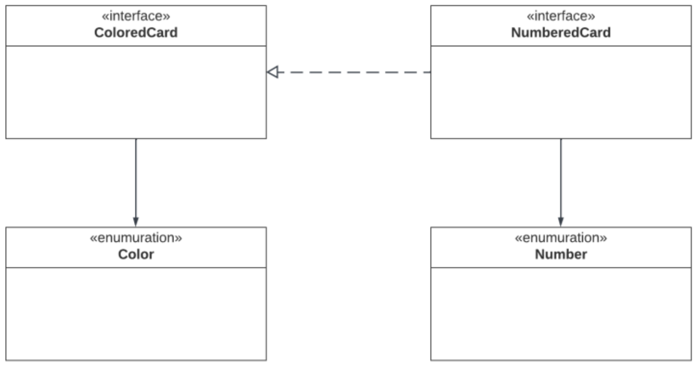
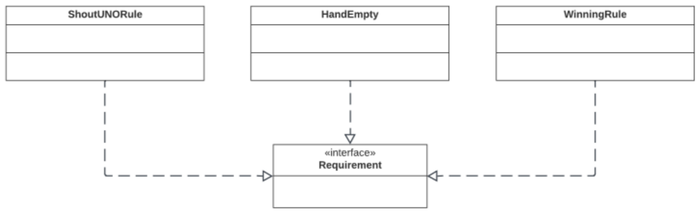


Figure 9: The Number and Color Enumerations

The Requirements Interface

The Requirements interface serves as a contract defining a set of requirements that need to be fulfilled in order to perform certain actions or apply specific rules within the game. It acts as a mechanism to encapsulate and generalize the conditions that must be met for an action to be valid or a rule to be applied.

By implementing the Requirements interface, these classes provide the necessary logic to evaluate whether the requirements are met or not. This design approach promotes extensibility and flexibility, allowing for the creation of additional requirement classes to support different game rules and actions. It also enables easy substitution of requirements by providing alternative implementations without affecting the existing codebase.



The Action Interface

The purpose of this interface is to define a common behavior for different types of actions that can be executed during the game. Figure (10) depicts the action interface.

Actions in the game are typically invoked after the rules of the game are checked and the requirements for those rules are satisfied. The Action interface allows for decoupling the specific actions from the rule implementation, making it flexible to add new actions or modify existing ones without affecting the rule logic.

When an action needs to be performed, the game would call the performAction() method on the appropriate Action object. This allows the action to access and interact with the game state and perform the necessary actions based on the game's current state.

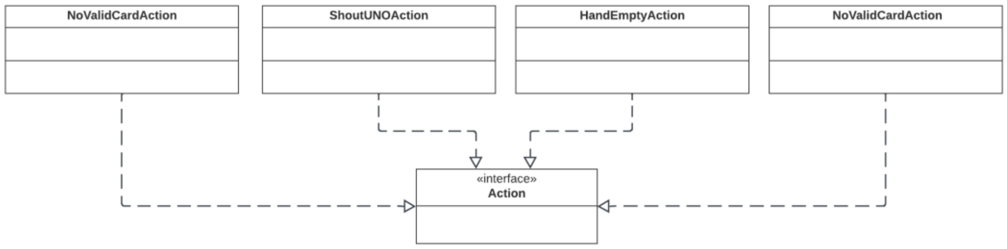


Figure 10: The Action Interface

IV. THE SOLID PRINCIPLES

Single Responsibility Principle (SRP)

The Single Responsibility Principle states that a class should have only one reason to change. In other words, a class should have a single responsibility or purpose and should be focused on performing that responsibility.

The proposed code adheres SRP for example:

- The Game class is responsible for managing the game logic and flow, including player turns, card validation, and executing actions. It encapsulates the core functionality of the game.

- The DeckFactory class is responsible for populating the deck with cards of various types and colors. It focuses solely on deck creation and initialization.

- The Hand class represents a player's hand and is responsible for managing the cards in the hand, drawing new cards from the deck, and playing cards. It has a single responsibility related to the player's hand management.

- The Referee class handles the validation of rules before and after playing a card. It ensures that the game follows specific rules and performs corresponding actions. It has a clear responsibility of rule validation.

- The Validator class is responsible for validating whether a card played by a player is valid according to the game rules. It ensures the correctness of card plays.

- The Executor class is responsible for executing the actions associated with specific cards. It maps each card to its corresponding action and performs the necessary actions when a valid card is played. The class's responsibility is limited to executing actions based on the card played, ensuring separation of concerns.

- The TurnManager manages the turn-based gameplay. It encapsulates the logic for determining the current player, advancing to the next turn, and handling turn-related actions. It doesn't handle unrelated concerns like player management or game rules

- The Leaderboard class manages the leaderboard and related score calculations. It encapsulates the logic for updating and calculating scores, as well as generating the leaderboard display. It doesn't handle other unrelated concerns like player management or game logic.

Open/Closed Principle (OCP)

The code shows how different types of cards, rules, and actions can be easily added without changing the existing code. It uses a technique called inheritance and polymorphism, which allows new cards to be created by building on the existing 'Card' class and implementing specific actions for each card.

Similarly, developers can create new rules by following a set of requirements and defining corresponding actions. This flexibility lets them customize the game's rules and behaviors to fit their specific needs.

The code also provides a way to create different dealers for the game. While a default dealer is already included, developers can easily implement their own dealer by following a standard interface. This makes it possible to have various dealer types in the game.

By using inheritance and polymorphism, the code becomes more modular and expandable. It allows for the addition of new features and customizations while keeping the existing code intact. This makes the game implementation more flexible and reusable.

Liskov Substitution Principle (LSP)

- Game Class
  The LSP is maintained as the Game class can work with any valid substitutions of the dependent classes without impacting the behavior.

- Card Hierarchy: The inheritance hierarchy of the Card class and its derived classes (NumberCard, ActionCard, WildCard, etc.) exhibits LSP compliance. Each derived class can be used interchangeably with the base Card class, preserving the behavior and semantics expected from a card. For example, a WildCard can be substituted for a Card wherever a card object is required.

- Interface Implementations: The program defines various interfaces (CardAction, Rule, Requirement, Dealer) that provide contracts for implementing specific behaviors. The derived classes that implement these interfaces, such as ReverseAction, SkipAction, ColorRequirement, and custom dealer implementations, adhere to LSP by faithfully fulfilling the contracts defined by the interfaces. This allows for substitutability of different implementations without affecting the overall functionality of the program.

- Polymorphic Usage: The program leverages polymorphism to interact with objects based on their base class or interface types. For example, the Game class can operate on any derived class of Card without needing to know the specific type. This promotes extensibility and flexibility, allowing new card types or rule implementations to be seamlessly incorporated into the game.

Interface Segregation Principle

- CardAction Interface: The CardAction interface defines a set of methods specifically related to card actions, such as playCard(), drawCard(), and discardCard(). By segregating these methods into a separate interface, the code allows clients, such as the Player class, to implement only the necessary methods for handling card actions without being burdened by unrelated methods.
- Requirements Interface: The Requirements interface provides methods like check() and fulfilled(), which are dedicated to checking specific requirements for game actions. This interface allows developers to implement requirement-specific logic without being forced to implement unrelated methods. Clients can choose to implement the Requirements interface selectively based on their game requirements, promoting a cohesive and tailored design.

- Action Interface: The Action interface defines methods related to game actions, such as performAction(). By separating these methods into an interface, the code ensures that clients can implement and handle game actions independently without being constrained by other unrelated methods. This allows for flexibility and modularity in implementing different game actions.

Dependency Injection Principle

The code utilizes principle injection in various classes, such as the Game, Player, LeaderBoard, and TurnManager classes. By passing dependencies through constructors, the code explicitly declares its dependencies, making them easily replaceable with alternative implementations. This promotes flexibility and enables different components to be injected into the classes, facilitating easier testing and modification.

V. DESIGN PATTERNS

The Strategy & Chain of Responsibility Design Pattern for Adding prePlayHandRules and postPreHandRules

In the design a combination of the "Strategy" design pattern and the "Chain of Responsibility" pattern has been utilized. This combination enhances the flexibility of the game engine, enabling developers to define their own rules and actions. The game engine determines whether the rules have been applied and executes the associated actions accordingly. By employing this design pattern combination, the game engine becomes modular and adaptable, accommodating a variety of rule configurations and actions.

The Strategy Pattern is used to encapsulate different rules and actions withing the game. The ‘Referee’ can dynamically select and execute different strategies (rules and actions) based on certain conditions. The ‘Requirement’ interface represents the strategy’s requirement, and the ‘Action’ interface represents the strategy’s action. The ‘beforePlayingRules’ and ‘afterPlayingRules’ are maps in the ‘Referee’ class which store the strategies.

The Chain of Responsibility pattern is used to execute the strategies sequentially. In the ‘testBeforePlaying ()’ and ‘testAfterPlaying ()’ methods of the ‘Referee’ class, the rules and actions are iterated over in a specific order, and each strategy is executed if the requirements are met. The ‘Requirement’ interface’s ‘checkRequirement ()’ method and the ‘Action’ interface’s ‘performAction ()’ method are called accordingly.

For detailed code implementation, please refer to Appendix B at the end of this report.

The Strategy Pattern for Executing the Actions of The Cards

In the provided code, the executable cards implement the CardAction interface, representing different actions. The Executor class utilizes the Strategy Pattern to determine the appropriate action to execute based on the card's name. By employing the Strategy pattern, new cards that implement the CardAction interface can be easily added to the Executor class, enhancing the code's flexibility and adaptability.

Please refer to Appendix C at the end of this report for the detailed code implementation.

The Factory Design Pattern used in DeckFactory class

The DeckFactory class is responsible for creating instances of the 'Card' class and adds them to the Deck. It serves as a centralized factory that encapsulates the creation logic for the different types of cards, such as numbered cards, action cards, wild cards. The 'populate()' method in the DeckFactory class demonstrates this creation process.

Please refer to Appendix D for detailed code implementation.

VI.CLEAN CODE PRINCIPLES

Clean code principles make sure that you have a well-structured, readable, and maintainable code that is easy to understand and modify. It focuses on making the code more understandable, reducing complexity, and minimizing potential for errors and bugs.

The provided code adheres to the clean code principles outlined by best bratices. Here are a few examples:

Naming

- Class names such as Game, DeckFactory, Hand, Referee, Validator, and Executor are clear and indicative of their purpose.
- Variable and method names, like 'currentHand', 'createPlayers()', and 'populate()', are descriptive and convey their intentions.

Functions/Methods

- The code is divided into methods, improving readability and maintainability. Functions and methods have clear and focused purposes, such as 'createPlayers()', responsible for creating game players, and 'setDealer()', responsible for setting the game's dealer.

Comments

- Well-placed comments effectively explain the purpose, behavior, and complex logic of certain components and algorithms. You can refer to Appendix E for examples of well-placed comments in the code.

Formatting

- The code follows consistent formatting conventions and maintains a clear and readable structure.
- Indentation and spacing are used consistently throughout the codebase, enhancing readability and maintainability.

Class Organization

- The classes in the codebase are organized and grouped based on their responsibilities.
- The Single Responsibility Principle and object-oriented design principles guide the class organization.

Error Handling

- The code implements proper error handling mechanisms, such as try-catch blocks, and provides meaningful error messages and fallback mechanisms. You can refer to Appendix F to explore specific examples of how errors were handled.

VII. EFFECTIVE JAVA PRINCIPLES

- Interface types to represent abstractions:
  The code makes good use of interface types (CardAction, Action, Requirement) to represent different behaviors and abstractions. This promotes modularity and allows for multiple implementations to fulfill those contracts. It enables loose coupling and provides flexibility to extend the code with new behavior implementations in the future.

- Composition over inheritance:
  The code demonstrates composition in various places. For example, the Wild and WildDrawFour classes could potentially be composed of a common WildCard implementation instead of directly extending the Card class. This design choice promotes flexibility, as it allows for easier modification and extension of behavior without being constrained by a rigid class hierarchy.

- Encapsulation and accessors:
  The code properly encapsulates fields by declaring them as private and provides appropriate accessors (get methods) to access their values. For example, the Skip class encapsulates the color field and provides a getColor() method to access it. Encapsulation ensures that the internal state of objects is protected, promoting data integrity and encapsulation benefits.

- Immutability
  While not all classes in the provided code are immutable, there are instances where immutability is applied. For example, the Skip class declares the color field as final, ensuring that it cannot be modified after initialization. Immutability provides benefits such as thread safety, simplicity, and prevention of unintended side effects.

- Naming conventions: The code follows the established Java naming conventions consistently. Class names start with an uppercase letter (Game, Card, LeaderBoard), interface names follow the same convention (ColoredCard, CardAction), and method names use camel case (performAction(), checkRequirement()). The code also uses meaningful and descriptive names for variables, promoting code readability.

CONCLUSIONS

A Java OOP Uno game engine is successfully designed, implemented and tested. The program fulfilled many design constraints, which may be used by other developers (i.e., not players) and the players may build their own variation of an Uno game.

The proposed solution created an abstract class, which is called ‘Game’ and consequently, a class is used to create a new game variation, where a developer only needs to create a new class that extends the abstract Game class.

The program is successfully designed such that Developers can choose from predefined rules when creating their own game variation and they can extend the code and add more rules to it.

The program also offers to the Developers ways to create new cards and new card-dealing mechanisms with minimal effort.

The program is engineered to have a class that is called GameDriver, which contains the main method. This main method contained two lines of code, one to instantiate a game object, and another one is used to invoke the play method.

V. REFERENCES

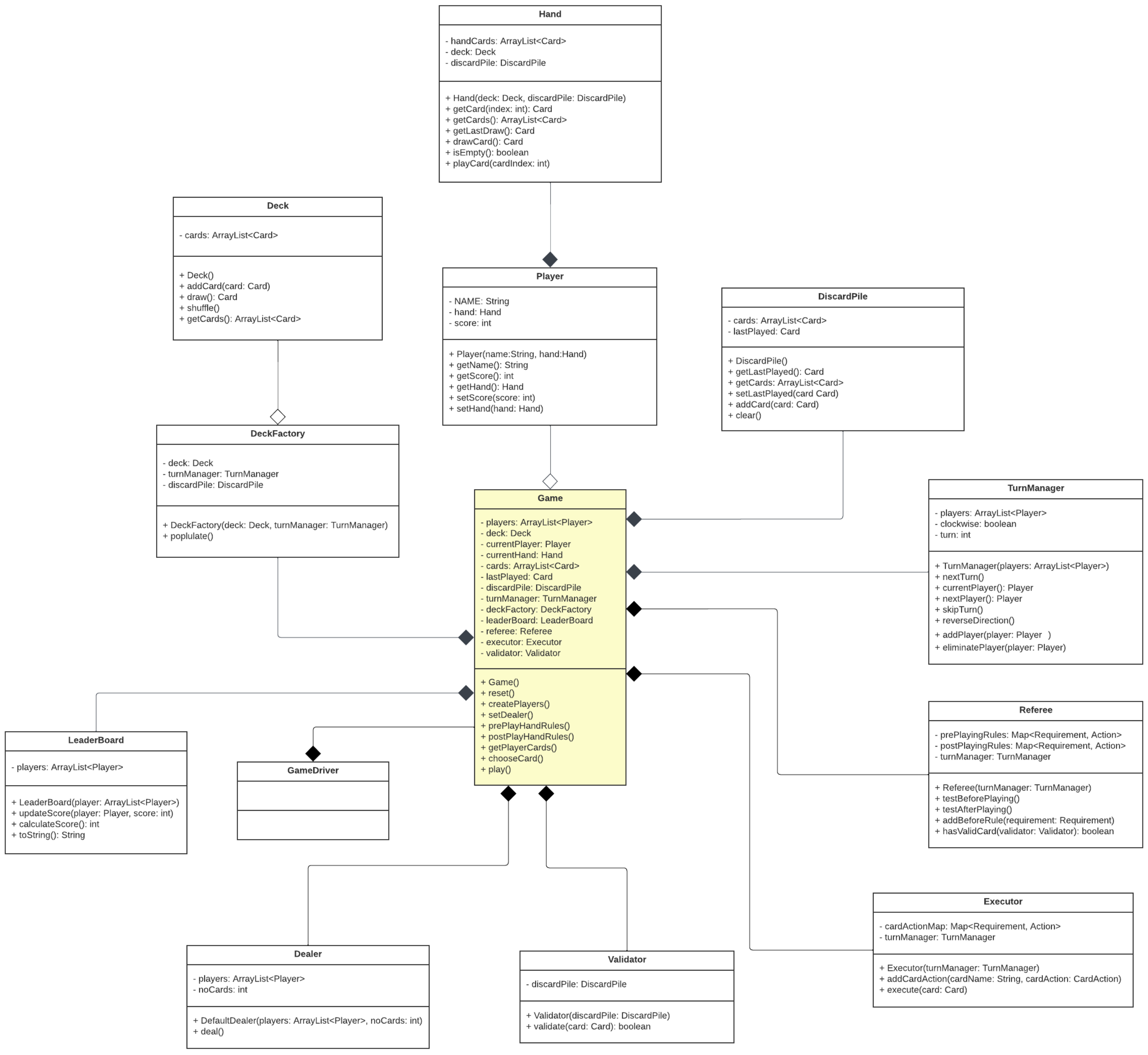
[1] www.WikiPedia.com / Playing cards / June 25, 2023

[2] Dr. Fahed Jubair, UNO Game Engine Assignment, Atypon Training Program 2023.

[2] Robert C.Martin, Clean Code: A Handbook of Agile Software Craftmanship.

VI. APPENDICES

Appendix A


Hand
- handCards: ArrayList<Card>
- deck: Deck
- discardPile: DiscardPile
+ Hand(deck: Deck, discardPile: DiscardPile)
+ getCard(index: int): Card
+ getCards(): ArrayList<Card>
+ getLastDraw(): Card
+ drawCard(): Card
+ isEmpty(): boolean
+ playCard(cardIndex: int)
Deck
- cards: ArrayList<Card>
+ Deck()
+ addCard(card: Card)
+ draw(): Card
+ shuffle()
+ getCards(): ArrayList<Card>
Player
- NAME: String
- hand: Hand
- score: int
+ Player(name:String, hand:Hand)
+ getName(): String
+ getScore(): int
+ getHand(): Hand
+ setScore(score: int)
+ setHand(hand: Hand)
DiscardPile
- cards: ArrayList<Card>
- lastPlayed: Card
+ DiscardPile()
+ getLastPlayed(): Card
+ getCards: ArrayList<Card>
+ setLastPlayed(card Card)
+ addCard(card: Card)
+ clear()
DeckFactory
- deck: Deck
- turnManager: TurnManager
- discardPile: DiscardPile
+ DeckFactory(deck: Deck, turnManager: TurnManager)
+ poplulate()
Game
- players: ArrayList<Player>
- deck: Deck
- currentPlayer: Player
- currentHand: Hand
- cards: ArrayList<Card>
- lastPlayed: Card
- discardPile: DiscardPile
- turnManager: TurnManager
- deckFactory: DeckFactory
- leaderBoard: LeaderBoard
- referee: Referee
- executor: Executor
- validator: Validator
+ Game()
+ reset()
+ createPlayers()
+ setDealer()
+ prePlayHandRules()
+ postPlayHandRules()
+ getPlayerCards()
+ chooseCard()
+ play()
TurnManager
- players: ArrayList<Player>
- clockwise: boolean
- turn: int
+ TurnManager(players: ArrayList<Player>)
+ nextTurn()
+ currentPlayer(): Player
+ nextPlayer(): Player
+ skipTurn()
+ reverseDirection()
+ addPlayer(player: Player)
+ eliminatePlayer(player: Player)
Referee
- prePlayingRules: Map<Requirement, Action>
- postPlayingRules: Map<Requirement, Action>
- turnManager: TurnManager
+ Referee(turnManager: TurnManager)
+ testBeforePlaying()
+ testAfterPlaying()
+ addBeforeRule(requirement: Requirement)
+ hasValidCard(validator: Validator): boolean
LeaderBoard
- players: ArrayList<Player>
+ LeaderBoard(player: ArrayList<Player>)
+ updateScore(player: Player, score: int)
+ calculateScore(): int
+ toString(): String
GameDriver
Executor
- cardActionMap: Map<Requirement, Action>
- turnManager: TurnManager
+ Executor(turnManager: TurnManager)
+ addCardAction(cardName: String, cardAction: CardAction)
+ execute(card: Card)
Dealer
- players: ArrayList<Player>
- noCards: int
+ DefaultDealer(players: ArrayList<Player>, noCards: int)
+ deal()
Validator
- discardPile: DiscardPile
+ Validator(discardPile: DiscardPile)
+ validate(card: Card): boolean

Appendix B

```
// Game Class
protected void prePlayHandRules(){

}

protected void postPlayHandRules(){
referee.addAfterRule(new ShoutUNORule(), new ShoutUNOAction());
referee.addAfterRule(new HandEmpty(), new
HandEmptyAction(this,leaderBoard));
referee.addAfterRule(new WinningRule(), new
WinningAction(this,leaderBoard));
}
```

```
// Referee Class
    private Map<Requirement, Action> prePlayingRules = new
HashMap<>();
    private Map<Requirement, Action> postPlayingRules = new HashMap<
>();

    public void testBeforePlaying(){

    for(Map.Entry<Requirement, Action> entry:
prePlayingRules.entrySet()){
       Requirement requirement = entry.getKey();
              Action action = entry.getValue();

              if(requirement.checkRequirement(turnManager)){
                 action.performAction(turnManager);
        }
      }
    }

    public void testAfterPlaying(){

    for(Map.Entry<Requirement, Action> entry:
postPlayingRules.entrySet()){
            Requirement requirement = entry.getKey();
            Action action = entry.getValue();

            if(requirement.checkRequirement(turnManager)){
                action.performAction(turnManager);
            }
        }
    }

    public void addBeforeRule(Requirement requirement, Action action){
        prePlayingRules.put(requirement, action);
    }

    public void addAfterRule(Requirement requirement, Action action){
        postPlayingRules.put(requirement, action);
    }
```

Appendix C

```
// Game Class
public void assignCardToAction(){
  executor.addCardAction("Draw2Action", new Draw2Action(turnManager));
  executor.addCardAction("Reverse", new ReverseAction(turnManager));
  executor.addCardAction("Skip", new SkipAction(turnManager));
  executor.addCardAction("Wild", new WildAction(discardPile));
  executor.addCardAction("WildDrawFour", new
DrawFourAction(discardPile, turnManager));
}


// Executor Class

private Map<String, CardAction> cardActionMap = new HashMap<>();

    private TurnManager turnManager;
    public Executor(TurnManager turnManager){
        this.turnManager = turnManager;
    }

    public void addCardAction(String cardName, CardAction cardAction){
        cardActionMap.put(cardName, cardAction);
    }
    public void execute(Card card){
        if(!(card instanceof ExecutableCard)){
            return;
        }
        CardAction cardAction = cardActionMap.get(card.getName());
        cardAction.action();
}
```

Appendix D

```
// The DeckFactory Class
public void populate(){

    // Adds numbered cards from 0-9 <first batch> to the deck
    for(Color color: Color.values()){
        for(Number number: Number.values()){
            deck.addCard(new NumberedCard(color, number));
        }
    }

    // Adds numbered cards from 1-9 <second batch> to the deck
    for(Color color: Color.values()){
        for(Number number: Number.values()){
            if(number.getNum() == 0){
                continue;
            }
            deck.addCard(new NumberedCard(color, number));
        }
    }

    // Adds action cards (Reverse, Skip, Draw Two) to the deck
    // We have two cards of each type for each color
    for(Color color: Color.values()){
        deck.addCard(new Reverse(color));
        deck.addCard(new Reverse(color));
        deck.addCard(new Skip(color));
        deck.addCard(new Skip(color));
        deck.addCard(new Draw2(color));
        deck.addCard(new Draw2(color));
    }

    for(int i = 0; i<4 ;i++){
        deck.addCard(new Wild(discardPile));
        deck.addCard(new WildDrawFour(discardPile, turnManager));
    }

    deck.shuffle();
}
```

Appendix E

```java
// Adds numbered cards from 0-9 <first batch> to the deck
for(Color color: Color.values()){
    for(Number number: Number.values()){
        deck.addCard(new NumberedCard(color, number));
    }
}

// Adds numbered cards from 1-9 <second batch> to the deck
for(Color color: Color.values()){
    for(Number number: Number.values()){
        if(number.getNum() == 0){
            continue;
        }
        deck.addCard(new NumberedCard(color, number));
    }
}

// Adds action cards (Reverse, Skip, Draw Two) to the deck
// We have two cards of each type for each color
for(Color color: Color.values()){
    deck.addCard(new Reverse(color));
    deck.addCard(new Reverse(color));
    deck.addCard(new Skip(color));
    deck.addCard(new Skip(color));
    deck.addCard(new Draw2(color));
    deck.addCard(new Draw2(color));
}
```

Appendix F

```
public Card getCard(int index){
        try{
            return cards.get(index);
        } catch (IndexOutOfBoundsException e){
            System.out.println("No such card - choose another");
            return null;
        }
}
```