

ATYPON



Instructors:
Fahed Jubair
Motasim Aldiab

done by:
Hamza Hassan

UNO GAME ENGINE

ABSTRACT

The main objective of the report is to create an Uno game engine that contains object-oriented programming and a powerful design to enable developers to make the Uno game and develop on it with the least possible effort.

Introduction

The purpose of this report is to provide an analysis of the Uno Game Engine code implemented in Java. The report will cover various aspects of the code, including object-oriented design, design patterns, adherence to clean code principles, compliance with "Effective Java" items, and alignment with SOLID principles. By evaluating these areas, we can gain insights into the quality and robustness of the codebase.

The Problem

I started thinking about my own design by analyzing the game, understanding it well, and studying the options available to me, in order to be the best possible image of the Uno game engine that the developer can make and extend it

Where I define everything that can be change and everything that can fixed

1. found that the rules of the game and the rules of the cards can be changed
2. The score calculation system may change
3. And a deck of playing cards may be added to some new cards or change their properties
4. the number of players is maybe 2-10
5. The number of cards a player can start with not fixed also

for this reason, must take all of this into account so that if any developer wants to change something in the game, he can do so with minimal effort and keep on the SOLID Principles.

So I took all these things into account when I designed my code but now let's talk about what I did step by step and discuss

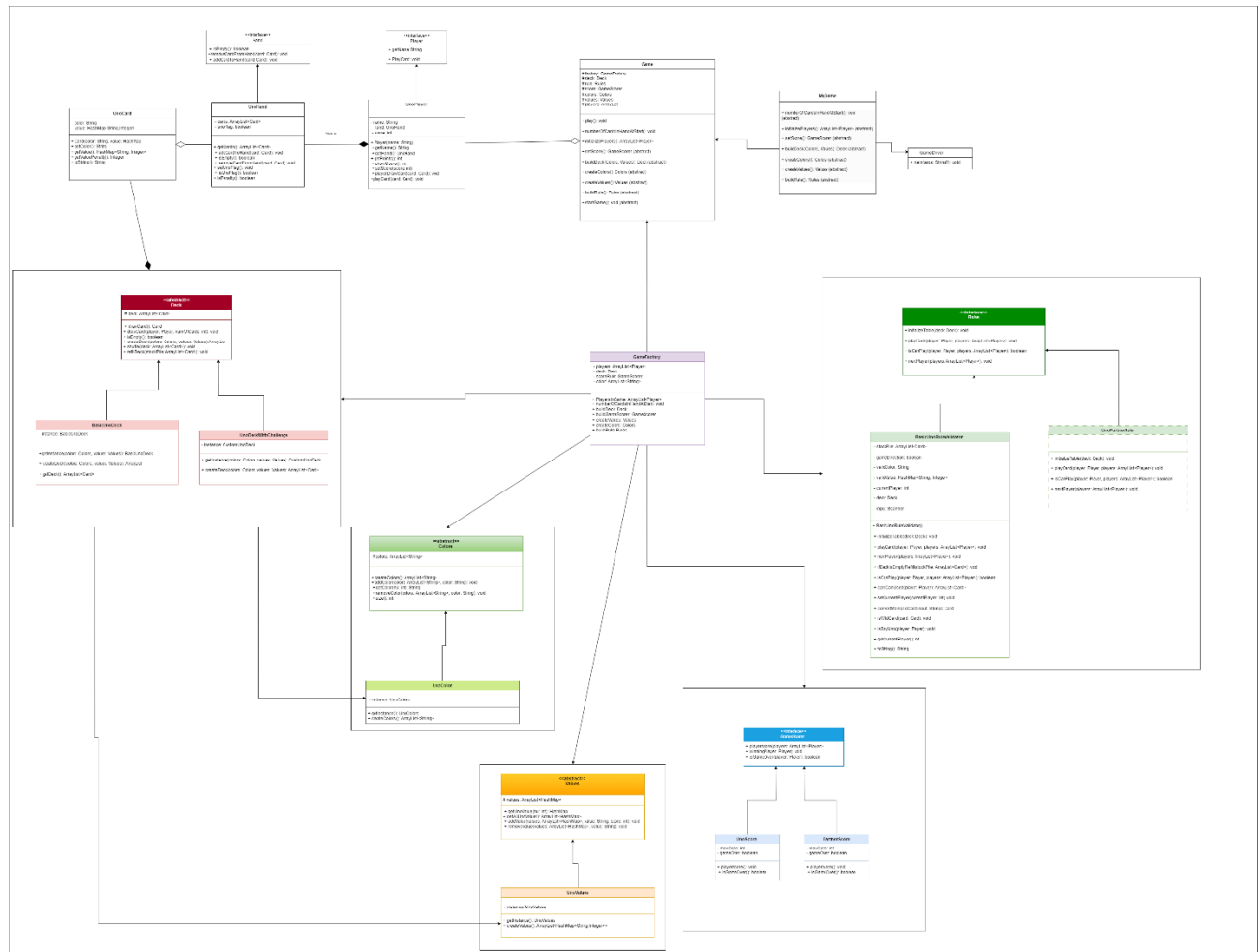
- object-oriented design,
- design patterns,
- clean code principles (Uncle Bob)
- "Effective Java" Items (Jushua Bloch)
- SOLID principles

for each class separately until we reached to the final design

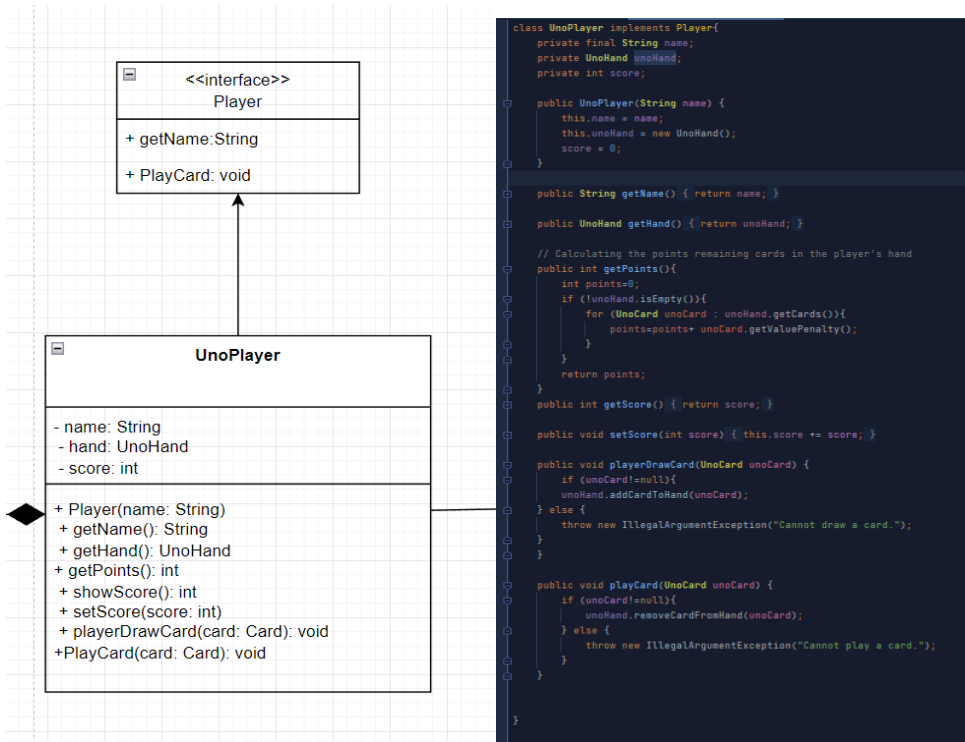
As in every Uno game, it should contain several main things like: Card, deck, player, Game rules, Score

SO my code contains many classes to simulate the game, and let's start by explaining them one by one and talking about the relationships between them:

What do we have?



1. UnoPlayer Class



1.1 object-oriented design,

`UnoPlayer`, which represents a player in the Uno game. Its responsibilities include managing the player's name, hand, and score.

The class implements the `Player` interface, Since I thought that I needed to expand my system and add a new game such as card game or any game other than UNO, to create a player for it more quickly and easily

the `UnoPlayer` class has a composition relationship with the `Hand` class. An instance of the `Hand` class is created as a private member variable within the `UnoPlayer` class. This composition relationship signifies that a player has a hand of Uno cards....

1.2 clean code principle

a. Meaningful Naming:

The code adheres to the principle of meaningful naming. The variable and method names are self-explanatory, concise, and follow standard naming conventions. For example, the variable "name" represents the player's name, and the method "getHand" retrieves the player's UnoHand.

b. Encapsulation:

The class effectively utilizes encapsulation by declaring instance variables as private and providing appropriate getter and setter methods. This encapsulation ensures data integrity and encapsulates the internal state of the UnoPlayer.

c. Exception Handling:

The code includes exception handling to address potential errors. In methods like "playerDrawCard" and "playCard," the code checks if the provided UnoCard is null and throws an `IllegalArgumentException`. This approach helps to handle invalid inputs and promotes robustness.

1.3 Effective Java

1) Item 2: Consider Builder Pattern for Complex Constructors:

The UnoPlayer class does not have a complex constructor, so it does not violate this item. The current constructor takes a single parameter, the player's name, which is sufficient for creating an instance.

2) Item 3: Enforce Singleton Property with a Private Constructor or an Enum Type:

The UnoPlayer class does not aim to be a singleton, so it does not conflict with this item. Each player in the game can have its own instance of the UnoPlayer class.

3) Item 15: Minimize Mutability:

The UnoPlayer class demonstrates appropriate immutability as the name, unoHand, and score variables are all declared as final. This choice ensures that once an instance of UnoPlayer is created, its state cannot be changed, preventing unexpected modifications.

1.4 SOLD principle.

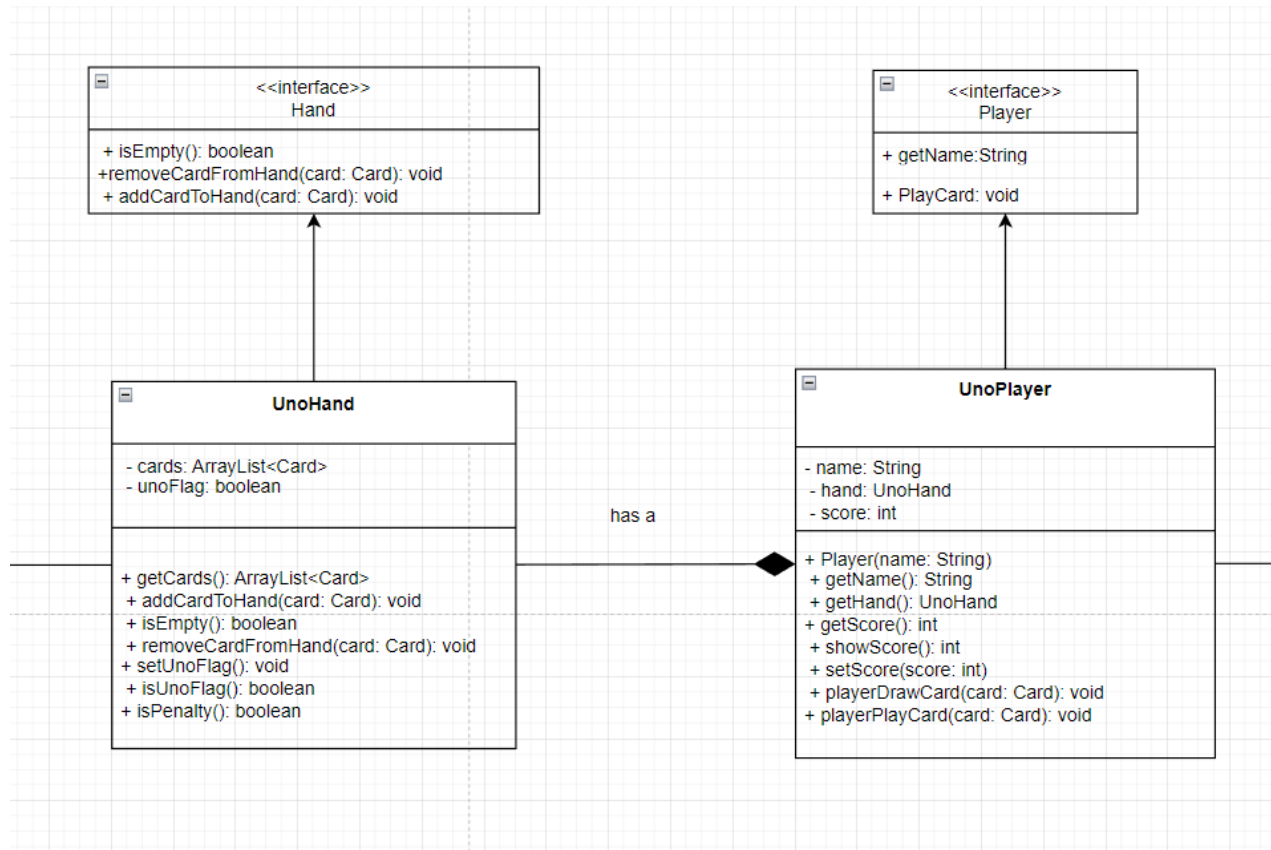
1. Single Responsibility Principle (SRP):

The UnoPlayer class appears to follow the SRP by focusing on managing player-specific functionality, such as maintaining the player's hand, calculating points, and handling card-related operations.

2. Open-Closed Principle (OCP):

The UnoPlayer class is relatively closed for modification, as its core functionality is well-defined and doesn't require frequent modifications.

2.Hand Class

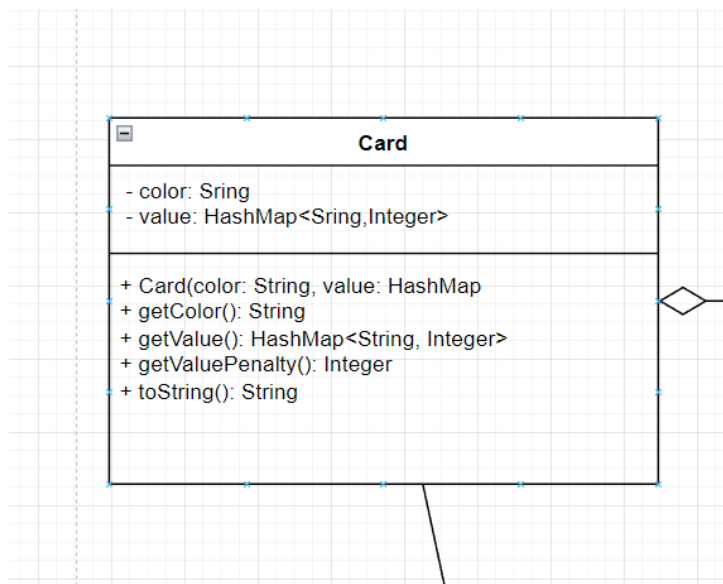


UnoHand Class, which represents a player's hand of Uno cards. Its responsibilities include managing the cards in the hand, tracking the Uno flag status, checking for penalties, and providing methods to get, add and remove cards.

class responsible for managing the player's hand.

To further improve the design and promote code reuse, I added an interface **Hand** that holds common behaviors because, as I mentioned earlier, if I needed to add a non-uno game, it would be easy to create a new hand class for it. This would allow for more abstraction and flexibility.

3.UnoCard class



1.1 object-oriented design,

UnoCard, which represents a card in the Uno game. Its responsibilities include managing to create card to game with colors and values.

1.2 clean code principle

i. Meaningful Naming:

The code adheres to the principle of meaningful naming. The variable and method names are self-explanatory, concise, and follow standard naming conventions.

ii. Encapsulation:

The class effectively utilizes encapsulation by declaring instance variables as private and providing appropriate getter and setter methods. This encapsulation ensures data integrity and encapsulates the internal state of the UnoCard.

1.3 Effective Java

1. Item 2: Consider Builder Pattern for Complex Constructors:

The UnoCard class does not have a complex constructor, so it does not violate this item. The current constructor takes a two parameter, color and value, which is sufficient for creating an instance.

1. Item 3: Enforce Singleton Property with a Private Constructor or an Enum Type:

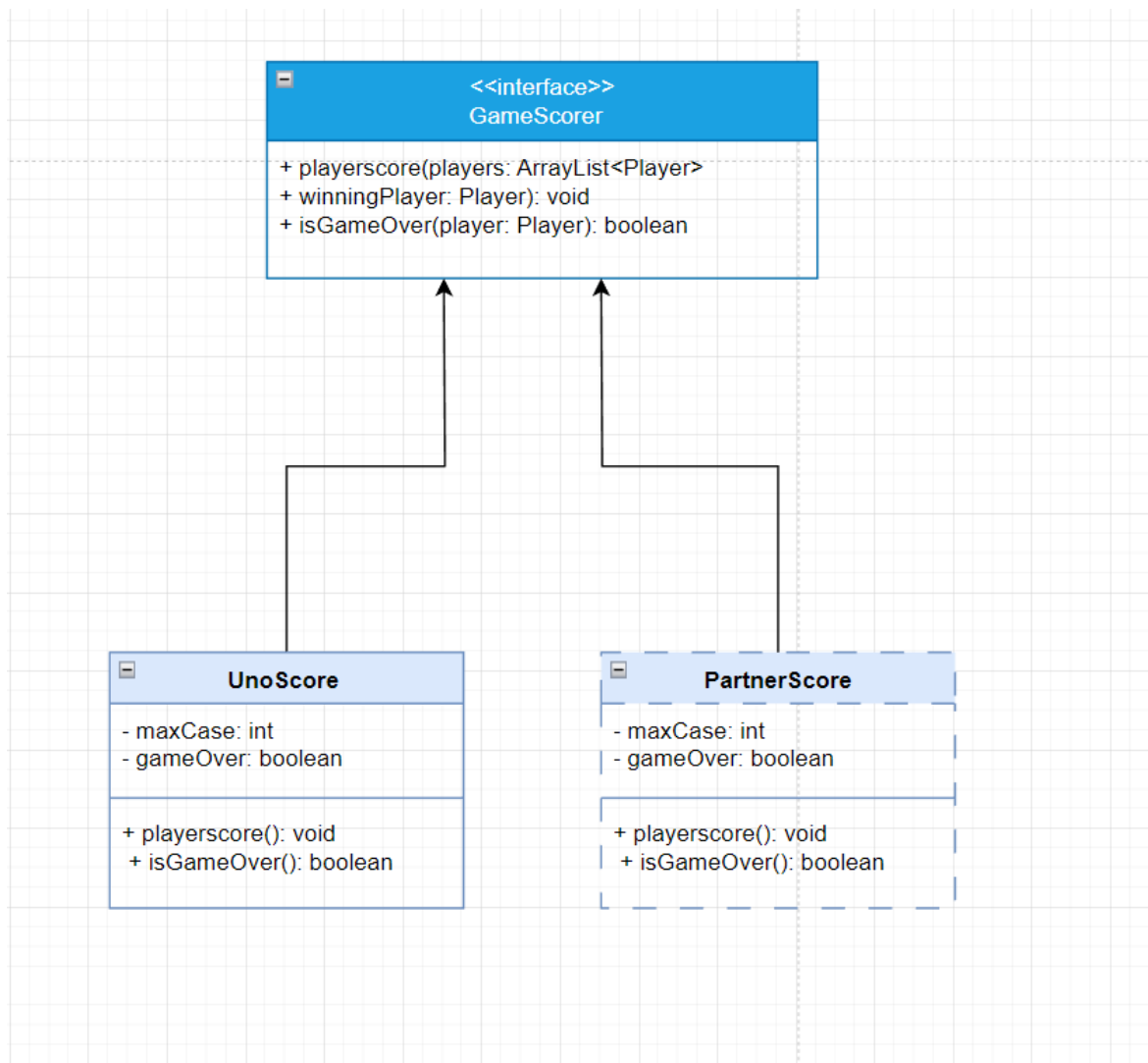
The UnoPlayer class does not aim to be a singleton, so it does not conflict with this item. Each card in the game can have its own instance of the UnoCard class.

1.4 SOLID principle.

1. Single Responsibility Principle (SRP):

The UnoCard class appears to follow the SRP by focusing on declare card functionality.

4.GameScorer



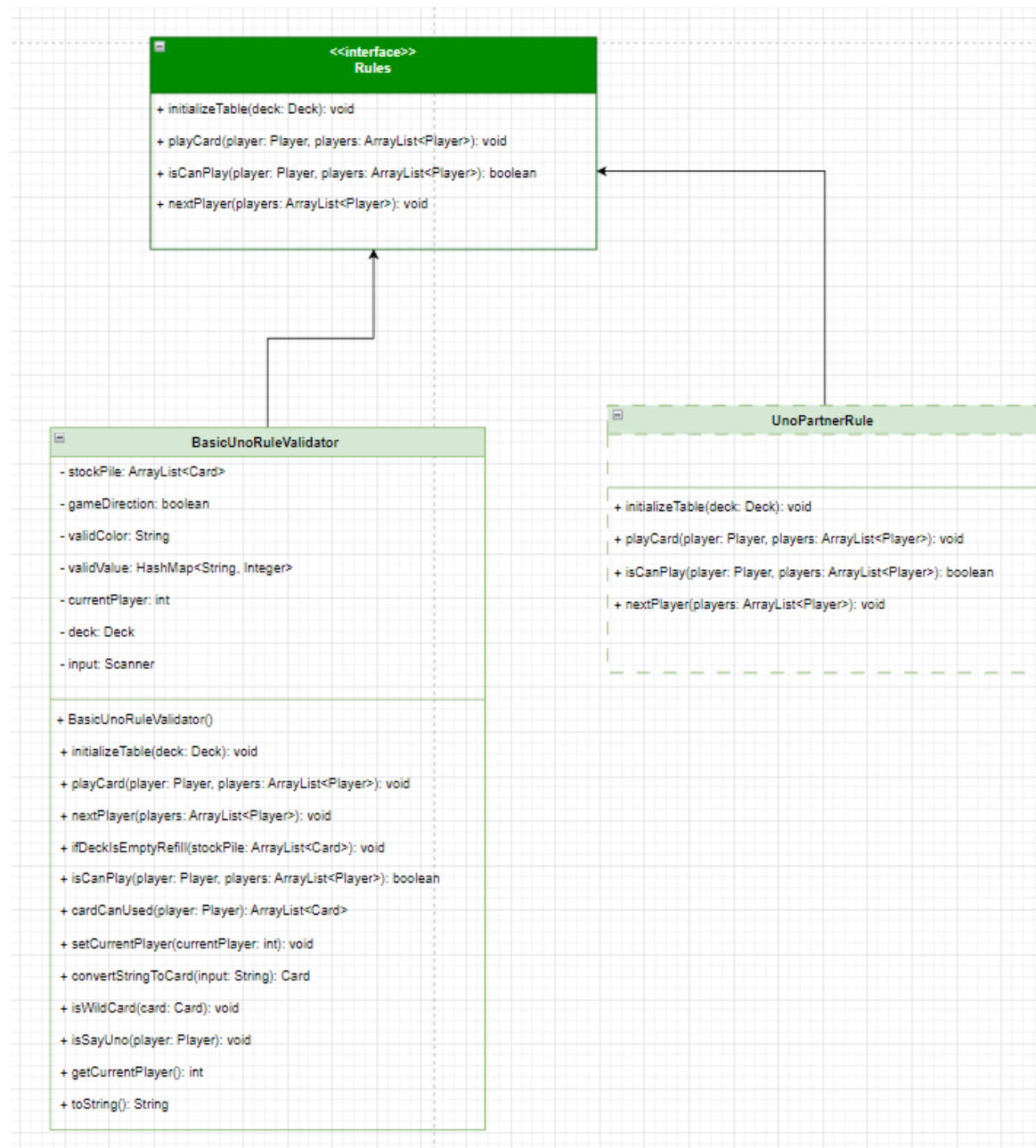
There may be several ways to calculate the score for each player and determine the winner of the game. The score may be calculated with a basic way like this:

- 20 points for each Draw 2, Reverse, or Skip card in an opponent's hand
- 50 points for Wild and Wild Draw 4 cards
- The face value for number cards (for example, an 8 card equals 8 points)

Or there may be another way for this.

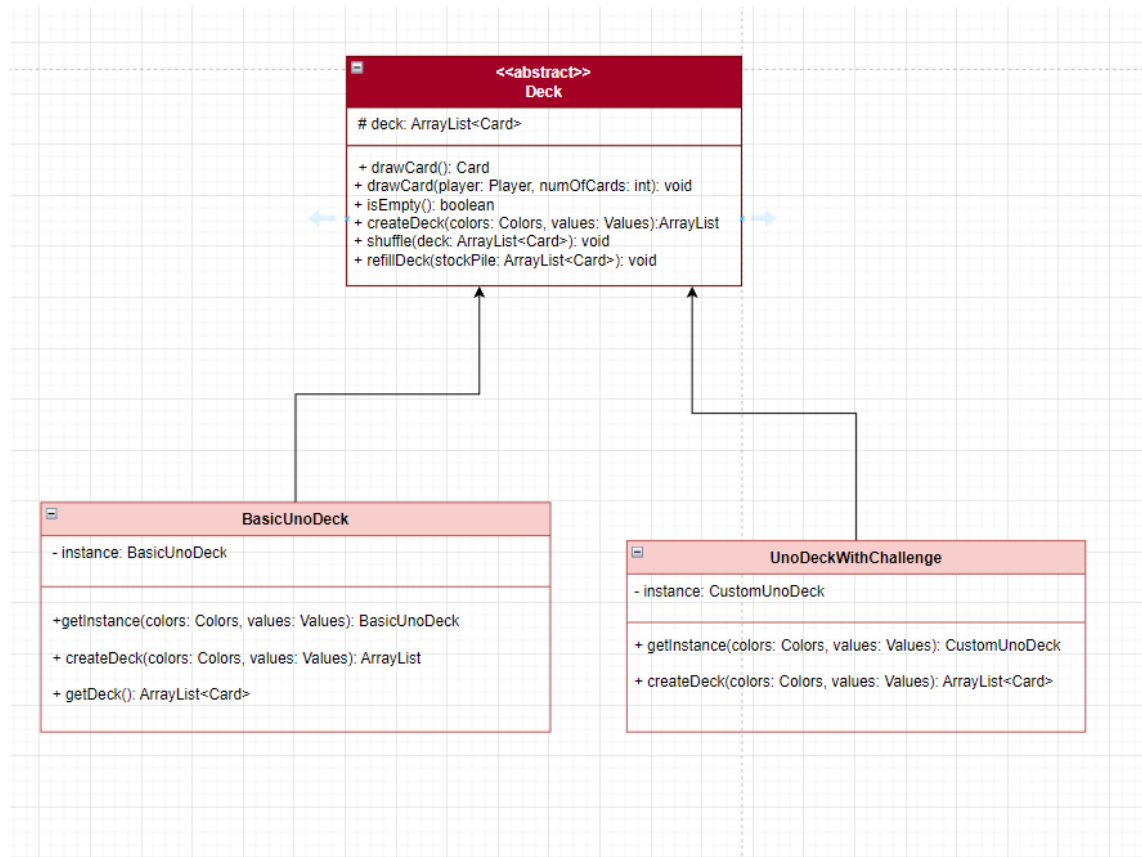
Because the Factory Method Pattern system was used to be able to easily add more than one method

5.Rules



One of the most important classes in Uno Game, as it contains many variable things that the developer may prefer to modify and use others. In this class, we will find such as the way to control the game cards and deal with them, as it determines the player's playing or preventing him and determines whose next , so I decided to put everything that is controlled by a specific rule Under this item, I added the possibility of inheritance so that the developer can create what he wants and use the Factory Method Pattern so that it is easy for the developer to add what he made to the system without modifying anything on it

6.Deck



4.1 Object-Oriented Design:

deck class

Developers may need to build a deck with different cards than the normal ones and add new cards or cancel some cards for that code demonstrates object-oriented design principles by utilizing classes and inheritance. the Deck and BasicUnoDeck classes handle the deck creation and management.

4.2 Design Patterns Used:

1. Singleton Pattern:

Because in the Uno game, you will only need one copy of the deck the **BasicUnoDeck** class employs the Singleton pattern by ensuring only one instance of the deck is created. The `getInstance()` method restricts the creation of multiple instances, promoting efficient resource utilization.

2. Factory Method Pattern:

The `createDeck()` method in the `BasicUnoDeck` class acts as a factory method. It provides a flexible way to create different types of decks while adhering to a common interface.

4.3 Clean Code Principles (Uncle Bob):

1. Meaningful Naming:

The code follows the principle of meaningful naming, with variables and methods having descriptive names that convey their purpose and functionality.

2. Single Responsibility Principle (SRP):

the `Deck` and `BasicUnoDeck` classes handle deck-related operations.

3. Exception Handling:

The code includes exception handling. For instance, the `drawCard()` method checks if the deck is empty before performing operations to prevent errors.

4.4 Effective Java Items (Joshua Bloch):

1. Singleton Property:

The `BasicUnoDeck` class implements the Singleton pattern to ensure only one instance is created, promoting efficient use of resources.

2. Inheritance:

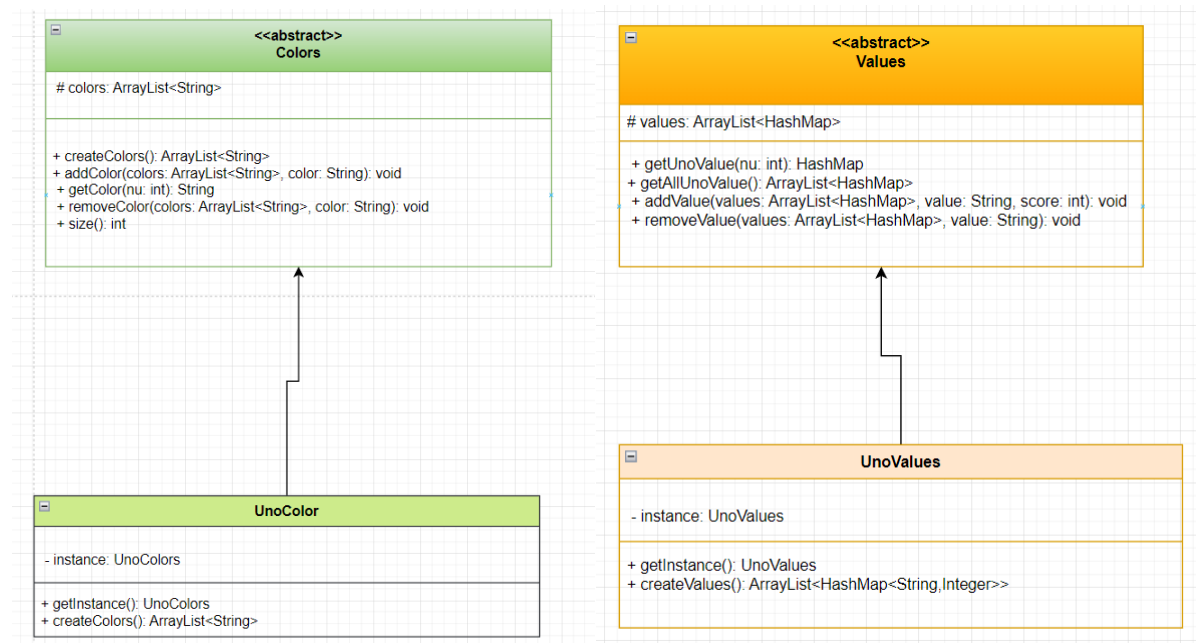
The `BasicUnoDeck` class extends the abstract `Deck` class, utilizing inheritance to provide a specialized implementation of the `createDeck()` method. This design allows for code reuse and flexibility in creating different types of decks.

4.5 SOLID Principles:

Open-Closed Principle (OCP):

The code demonstrates limited adherence to the OCP. While the classes are closed for modification, the Deck class could be extended or composed to allow for different types of decks without modifying the existing code.

7.Colors Class and Values Class



I have created the UnoColor class and the UnoValue class to represent the colors and values of cards in the game of Uno. This allows me to create cards specifically for Uno with the colors and values that are known to us. Additionally, I have implemented the ability to add, remove, or obtain these values and colors for Uno cards.

when the developer need to create new deak with new card can use the UnoColors and UnoValues then and the new color and new value the create the new card

like i do in DeckWithChallenges class:

```
import java.util.ArrayList;

public class DeckWithChallenges extends Deck {
    private static DeckWithChallenges instance;

    private DeckWithChallenges(UnoColors unoColors, UnoValues unoValues) {
        deck=createDeck(unoColors, unoValues);
        shuffle(deck);
    }

    public static DeckWithChallenges getInstance(UnoColors unoColors, UnoValues unoValues) {
        if (instance == null) {
            instance = new DeckWithChallenges(unoColors, unoValues);
        }
        return instance;
    }

    public ArrayList<UnoCard> createDeck(UnoColors unoColors, UnoValues unoValues) {
        // Add new color
        ArrayList<UnoCard> deck=BasicUnoDeck.getInstance(unoColors, unoValues).getDeck();
        // Add new 3 value
        UnoColors.addColor(UnoColors.colors, color: "Purple");
        UnoValues.addValue(UnoValues.values, value: "You should do push-ups 10 times", score: 0);
        UnoValues.addValue(UnoValues.values, value: "You should do 20 jumps", score: 0);
        UnoValues.addValue(UnoValues.values, value: "You must not speak until the end of the game", score: 0);
        //create 12 new card using new color and new values and add it to the uno
        for (int i = 0; i < 4; i++) {
            deck.add(new UnoCard(UnoColors.getUnoColor( i% 5), UnoValues.getUnoValue( i% 16)));
            deck.add(new UnoCard(UnoColors.getUnoColor( i% 5), UnoValues.getUnoValue( i% 17)));
            deck.add(new UnoCard(UnoColors.getUnoColor( i% 5), UnoValues.getUnoValue( i% 18)));
        }
        return deck;
    }
}
```

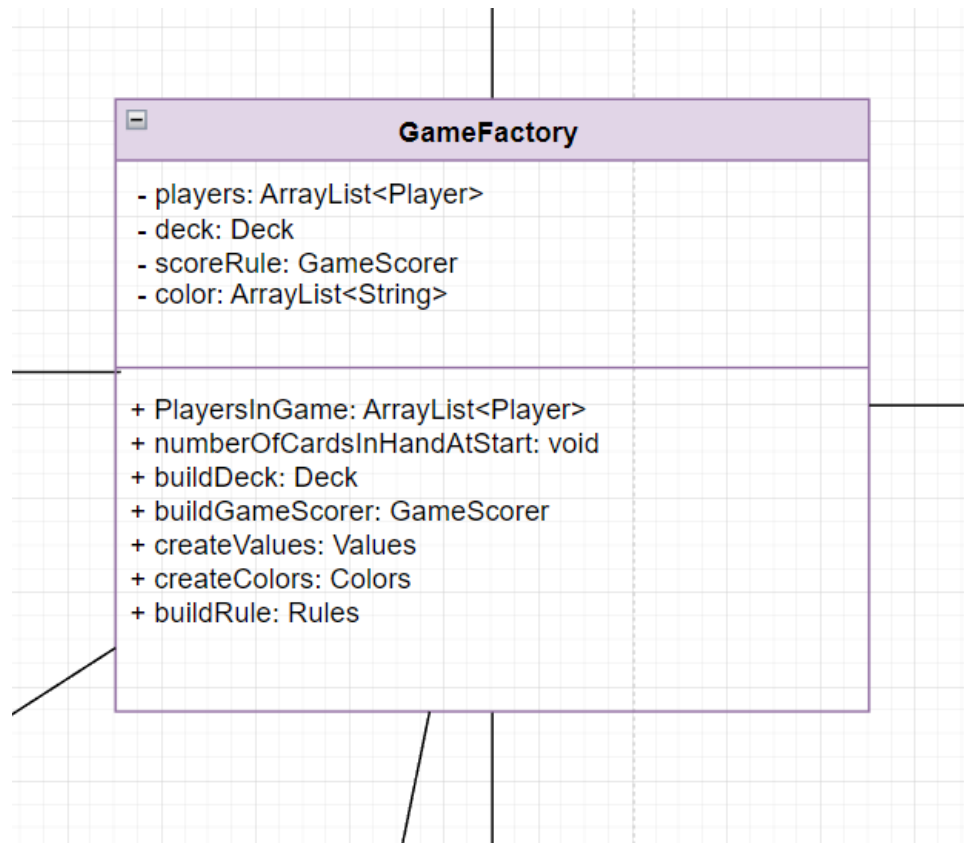
However, there may be instances where I need to create cards that are not meant for Uno, but for a different game altogether. To accommodate this requirement and allow for future expansion and inclusion of new games, I have designed an abstract class called Colors and put into it the common thing will needed. This class enables the flexibility of the system to support the creation of entirely different cards for various games, not limited to Uno

design patterns:

Singleton Pattern: The UnoColors class implements the Singleton pattern, ensuring that only one instance of the class can be created.

Factory Method Pattern: If the game is not UNO and you need to build and choose a set of colors and other values

8.GameFactory Class



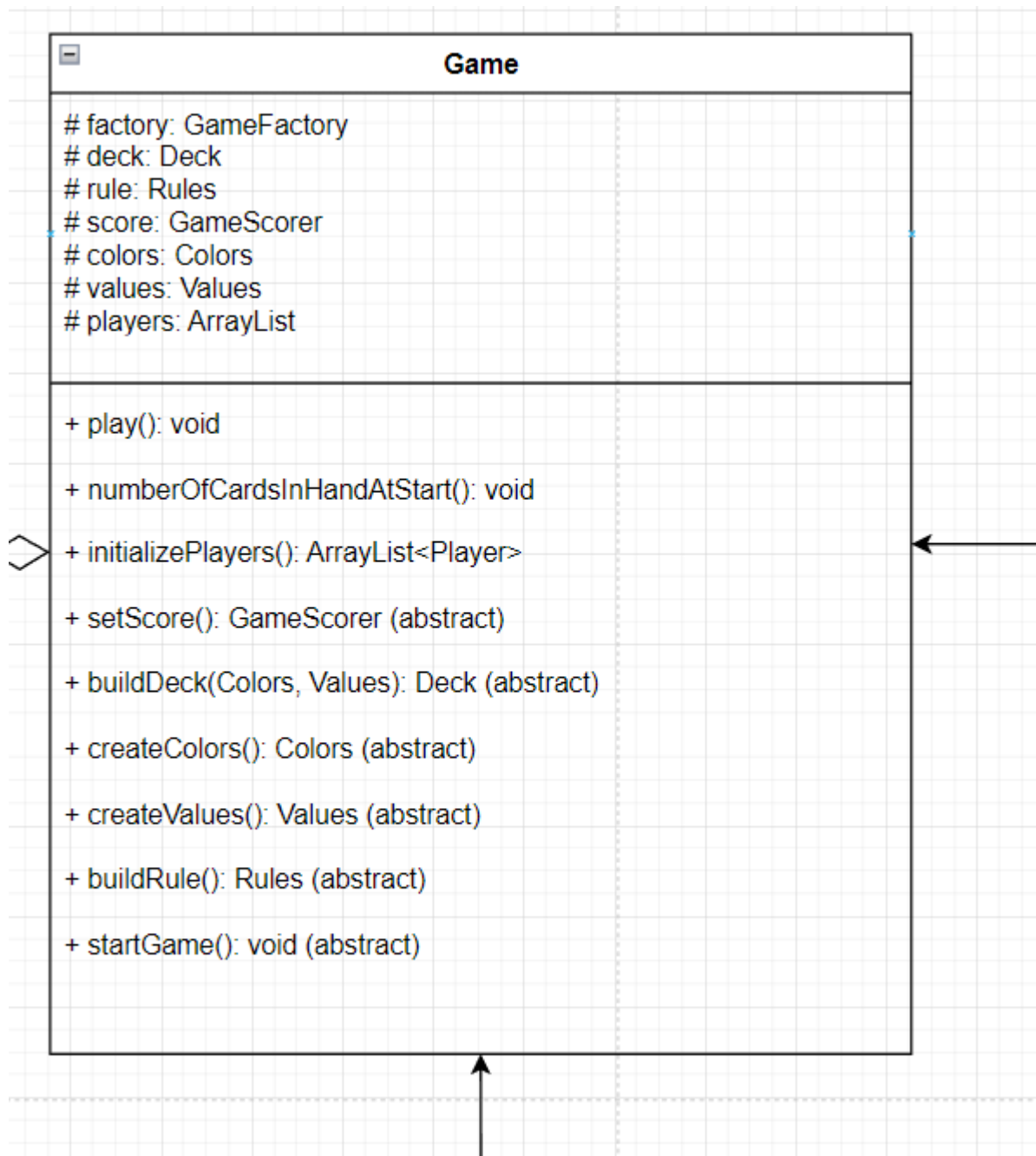
The GameFactory class acts as a factory for setup all components game need.

GameFactory class follows the factory pattern by providing methods for creating instances of different game components. It abstracts the process of object creation and allows the flexibility to switch between different implementations.

Single Responsibility Principle (SRP): The GameFactory class has a single responsibility of creating game components. Each method within the class is focused on a specific task, such as creating players, decks, scorers, values, colors, and rules, adhering to the SRP.

Open/Closed Principle (OCP): The GameFactory class is designed to be extensible by providing options for different implementations of game components through method parameters. It can easily accommodate new types of decks, scorers, values, colors, and rules by adding additional cases to the switch statements.

9.GameFactory Class



To reduce the effort on the developer, when he does extend Game Class, he will be able to build any game that he wants, whether it is Uno or something else. All that the developer needs to create his own game is to build the abstract method using instance of factory game

And build a startGame method

where should be determine the number of players in game

And determine the number of cards that the player will start with

Then must initialized the table for play

and finally he should build the way is presented the game to the users

in Game Class I have method called [Play](#) that follows the template design method

```
// template method
public final void play() throws IllegalAccessException {
    unoColors = createColors();
    unoValues = createValues();
    deck = buildDeck(unoColors, unoValues);
    score = setScore();
    rule = buildRule();
    startGame();
}
```

It will be responsible for simulating the game, as any card game in the world is originally following the same system as it is

- You must provide a color and values for the cards
- Then we will have a Deck that we will play with it
- And the rules on which we will play must be available
- We set a score that will determine when the game ends and determine the winner
- Then the start Game

10. Final class will create by developer

this sample of how can be created

```
Game.java x MyGame.java x DeckWithChallenges.java x UnoValues.java x Values.java x
1 public class MyGame extends Game {
2     public MyGame() throws IllegalAccessException {
3         super();
4     }
5
6     @Override
7     public GameScorer setScore() {
8         return factory.buildGameScorer(1);
9     }
10
11     @Override
12     public Deck buildDeck(UnoColors unoColors, UnoValues unoValues) {
13         return factory.buildDeck(1, unoColors, unoValues);
14     }
15
16     @Override
17     public UnoColors createColors() {
18         return factory.createColors(1);
19     }
20
21     @Override
22     public UnoValues createValues() {
23         return factory.createValues(1);
24     }
25
26     @Override
27     public Rules buildRule() {
28         return factory.buildRule(1);
29     }
30 }
```

```
@Override
public void startGame() {
    System.out.println("Welcome to the UNO game.");
    PlayersInGame( numOfPlayers: 4);
    numberOfCardsInHandAtStart( numOfCards: 7);
    rule.initializeTable(deck);
    while (true) {
        UnoPlayer unoPlayer = unoPlayers.get(rule.getCurrentPlayer());

        if (rule.isCanPlay(unoPlayers)) {

            System.out.println("\nIt's " + unoPlayer.getName() + " turn to play, and the cards he has are: ----- ");
            for (UnoCard unoCard : unoPlayer.getHand().getCards()) {
                System.out.print(unoCard.toString() + " ");
            }

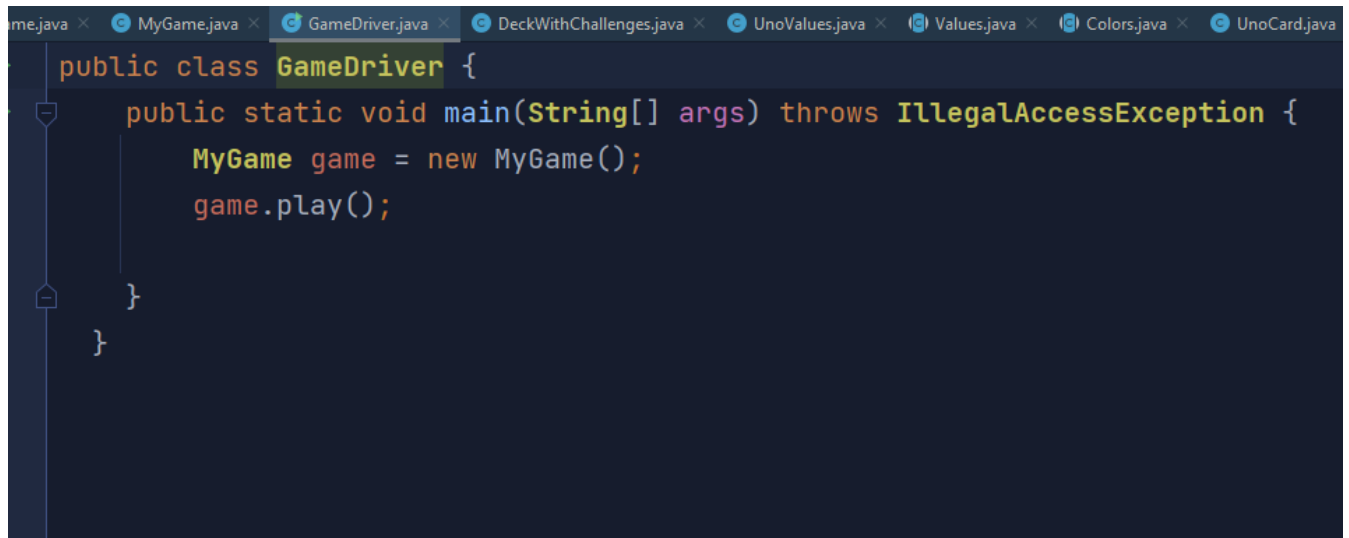
            System.out.println("\nthe cards valid to play :");
            for (UnoCard unoCard : rule.cardCanUsed(unoPlayer)) {
                System.out.print(unoCard.toString() + " ");
            }

            System.out.println("\nPLEASE choose one from them: ");
            rule.playCard(unoPlayers);

            if (unoPlayer.getHand().isEmpty()) {
                // This means that one of the players wins and the round ends Players' points are calculated and put into the winner's score
                score.playerscore(unoPlayers, unoPlayer);
                if (score.isGameOver(unoPlayer)) {
                    System.out.println("When the game is over\n" +
                        "Congratulations winner is " + unoPlayer.getName());
                    System.exit( STATUS: 0);
                }
            }
        }
        rule.nextPlayer(unoPlayers);
    }
}
```

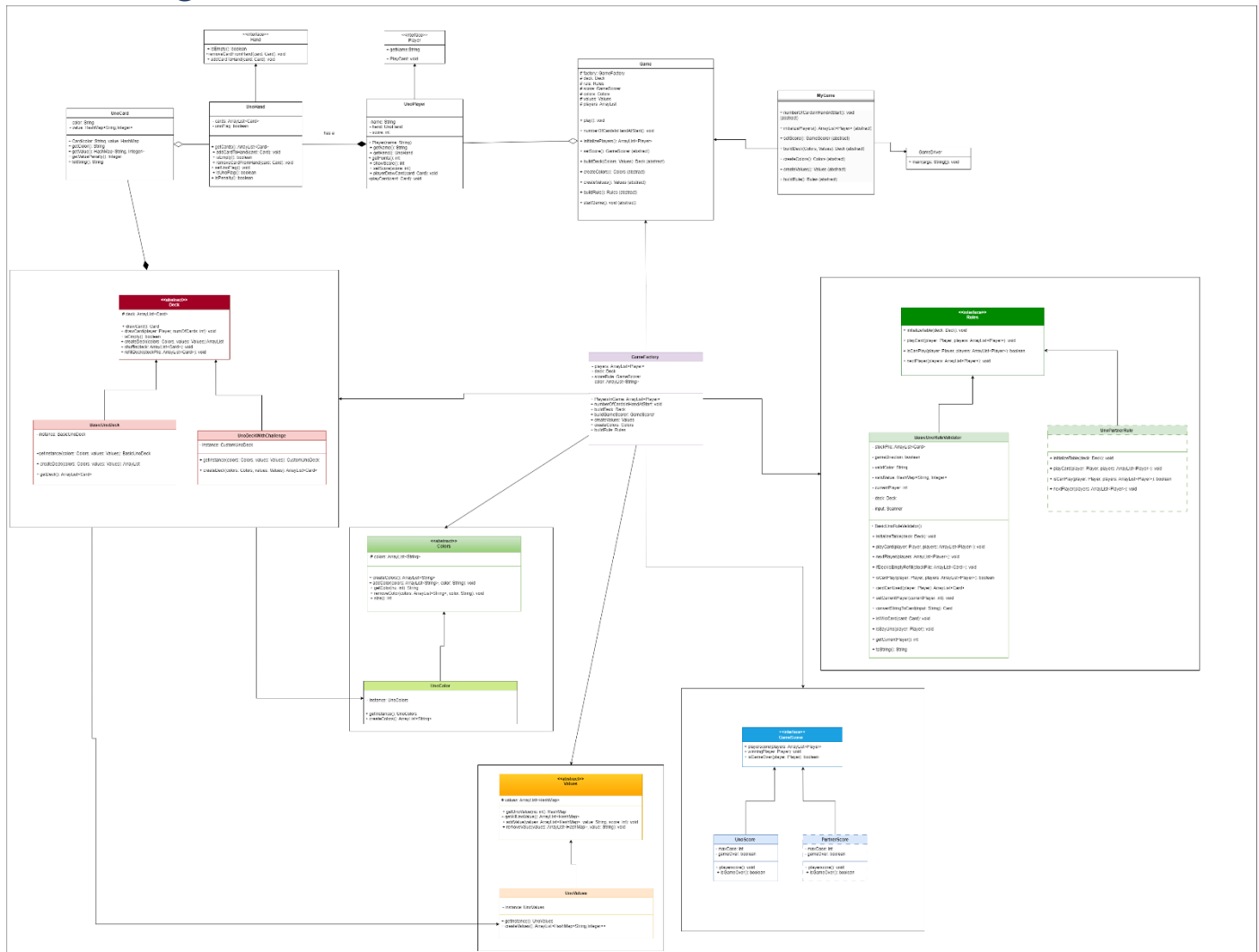
as simple as that.

11. And the unit test will be



```
me.java x MyGame.java x GameDriver.java x DeckWithChallenges.java x UnoValues.java x Values.java x Colors.java x UnoCard.java  
public class GameDriver {  
    public static void main(String[] args) throws IllegalAccessException {  
        MyGame game = new MyGame();  
        game.play();  
    }  
}
```

Final design



The end