Uno Game Engine Report

By Hamza Alhalabi

Table of Contents

Introduction

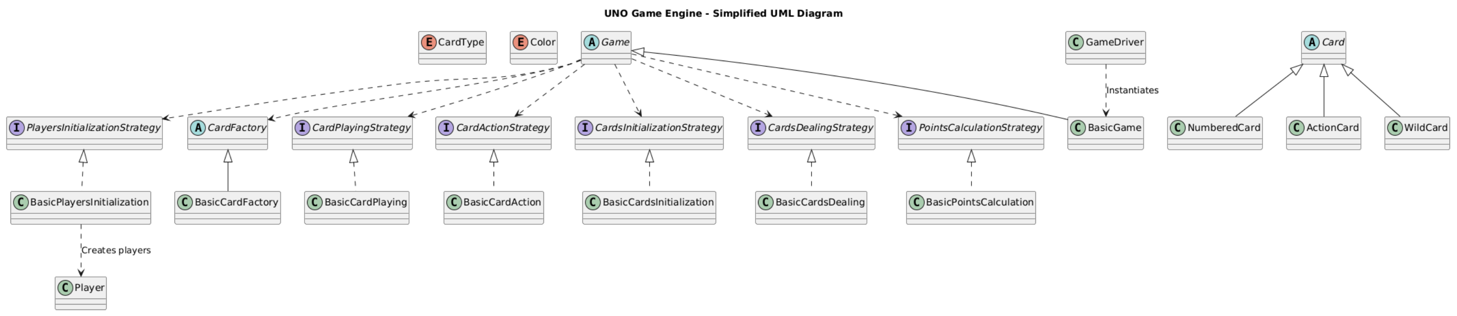
The project that I built is Uno Game Engine. It's designed for developers so they can extend the code and build their own variations of the game. I was eager to keep the design extensible and easy to understand by others.

I tried to applied the theoretical concepts in a real application, I used the concept of abstract and concrete classes effectively, I used more than three design patterns in my code and tried to commit to clean code principles as much as I could.

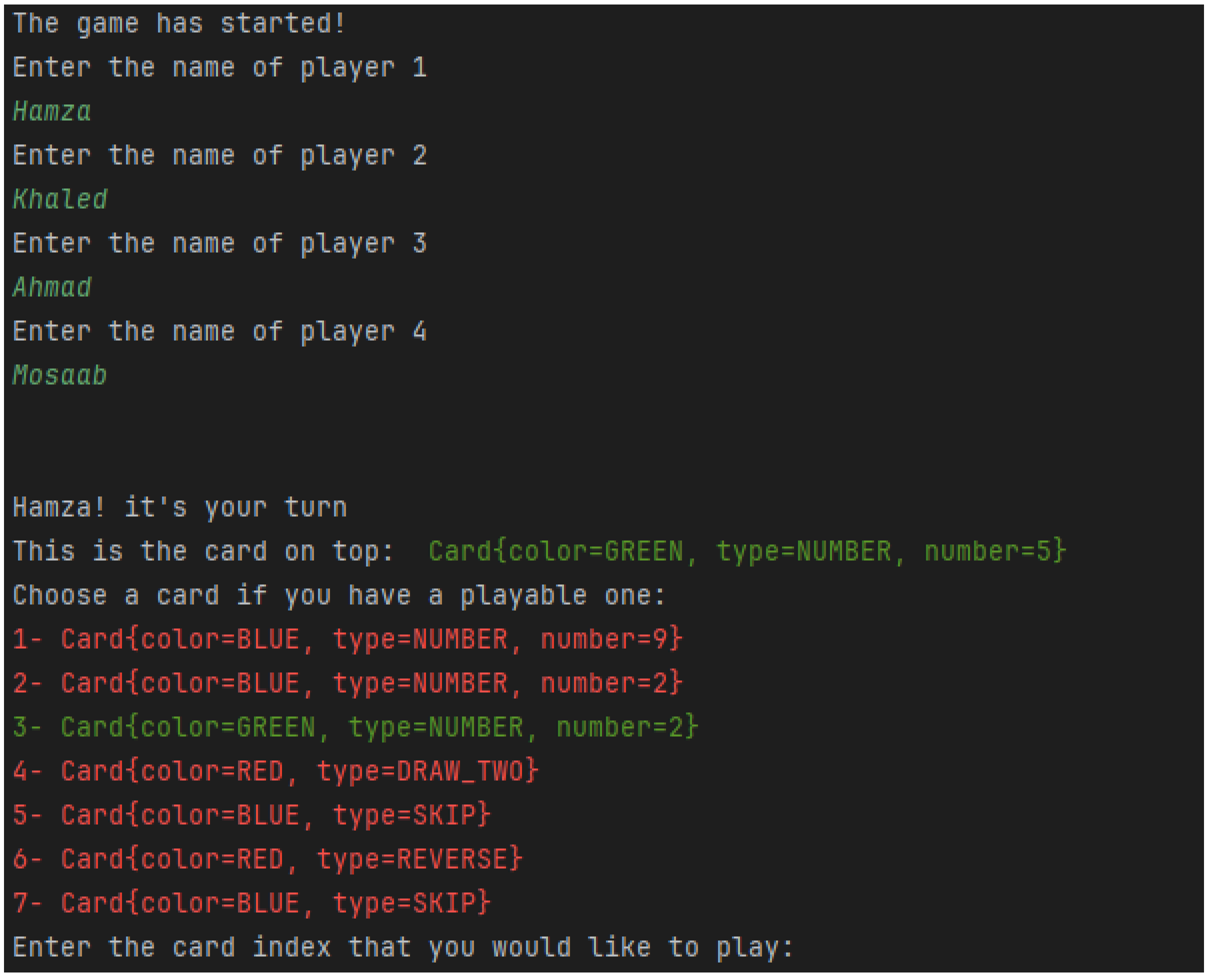
This project is not for developers only, it's ready to use by players to play as I added a set of basic rules that are known as the official rules worldwide.

At the beginning, I spent some time reading about the official rules of Uno game and different variations of it. I ignored the variations that use different cards or different rules totally like Uno-Flip and Uno-Attack. I also supposed that card types and colors are static between all variations, so we can't create a new variation with a purple or orange card! Developers are only able to change the number of cards for each type of cards.

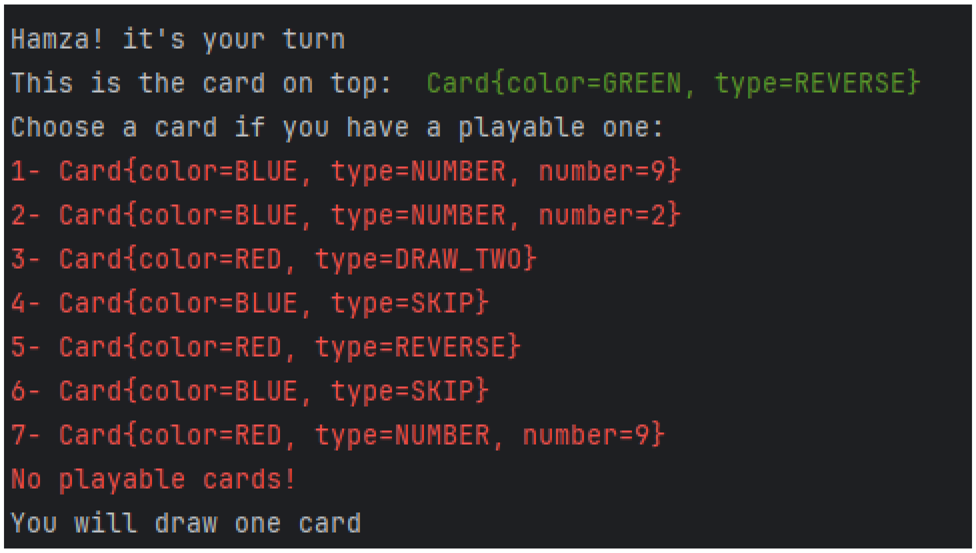
This is the simplified **UML** diagram for the project:

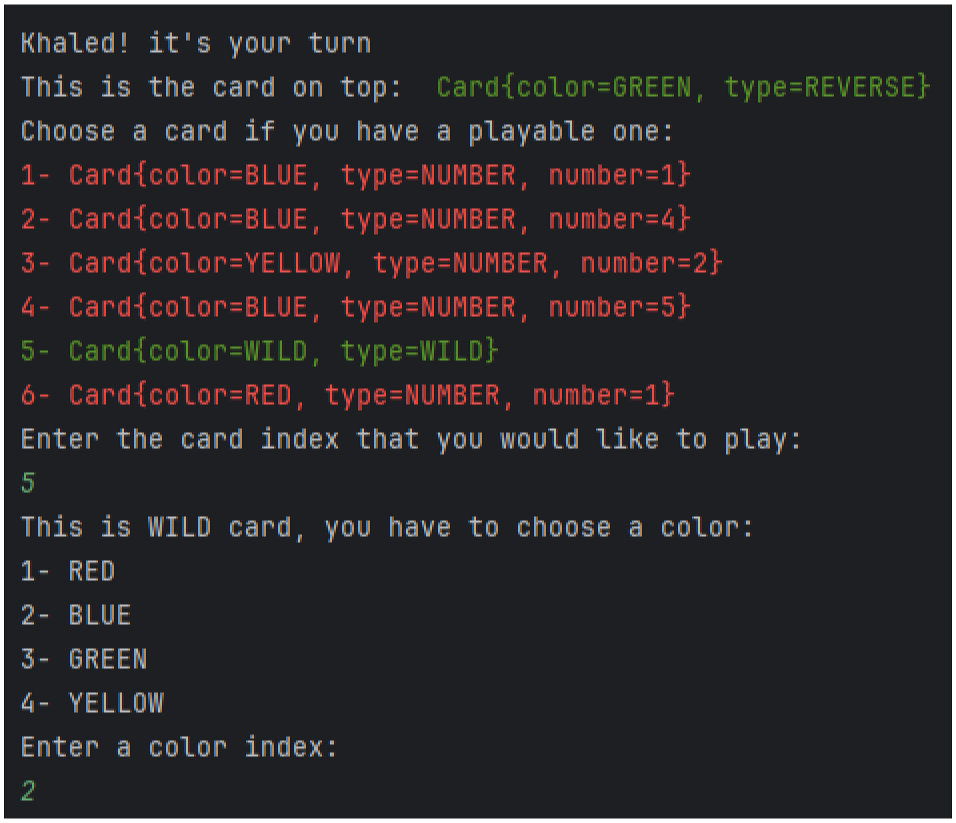


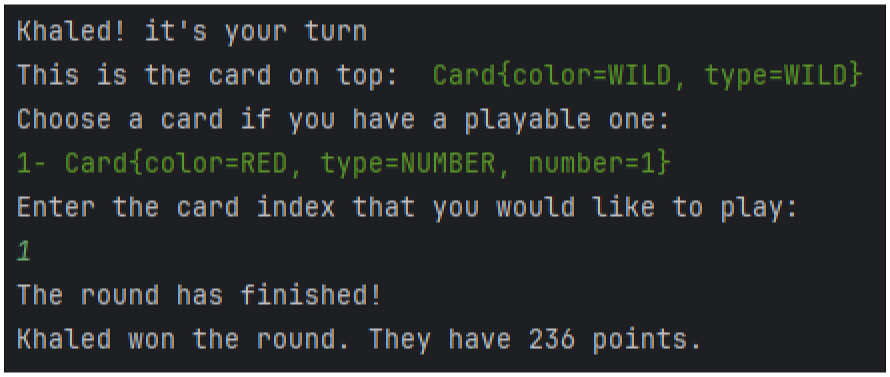
These are some snapshots on how the program looks like when running, the single green colored card is the card on top, and the list of player's cards is divided into **playable** cards in green, and **not playable** cards in red

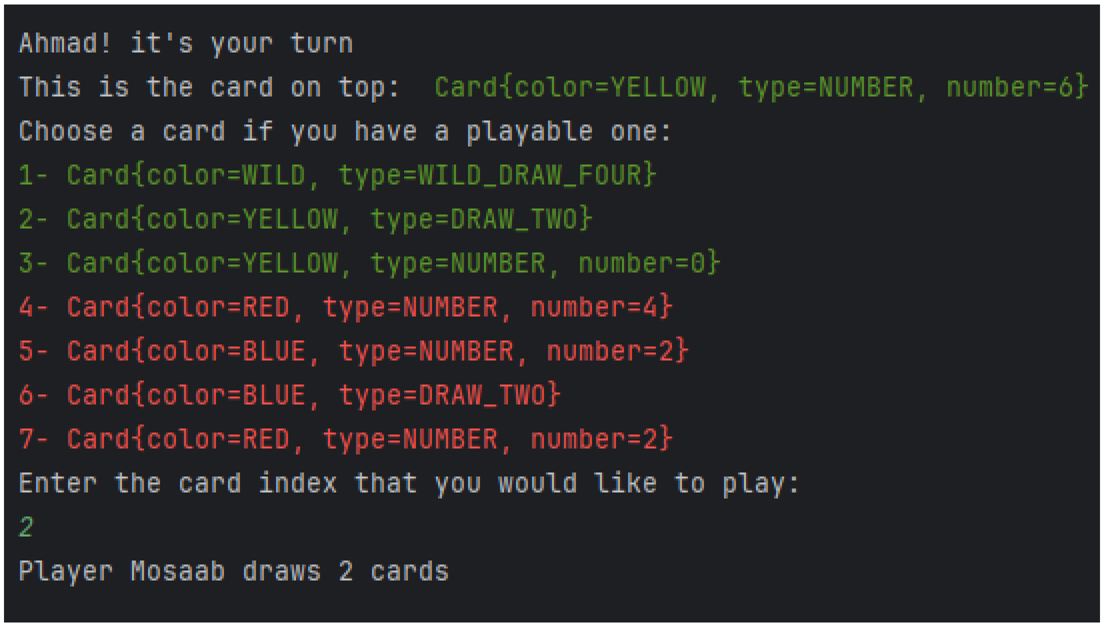


```
The game has started!
Enter the name of player 1
Hamza
Enter the name of player 2
Khaled
Enter the name of player 3
Ahmad
Enter the name of player 4
Mosaab


Hamza! it's your turn
This is the card on top:  Card{color=GREEN, type=NUMBER, number=5}
Choose a card if you have a playable one:
1- Card{color=BLUE, type=NUMBER, number=9}
2- Card{color=BLUE, type=NUMBER, number=2}
3- Card{color=GREEN, type=NUMBER, number=2}
4- Card{color=RED, type=DRAW_TWO}
5- Card{color=BLUE, type=SKIP}
6- Card{color=RED, type=REVERSE}
7- Card{color=BLUE, type=SKIP}
Enter the card index that you would like to play:
```


```
Hamza! it's your turn
This is the card on top:  Card{color=GREEN, type=REVERSE}
Choose a card if you have a playable one:
1- Card{color=BLUE, type=NUMBER, number=9}
2- Card{color=BLUE, type=NUMBER, number=2}
3- Card{color=RED, type=DRAW_TWO}
4- Card{color=BLUE, type=SKIP}
5- Card{color=RED, type=REVERSE}
6- Card{color=BLUE, type=SKIP}
7- Card{color=RED, type=NUMBER, number=9}
No playable cards!
You will draw one card
```


```
Khaled! it's your turn
This is the card on top:  Card{color=GREEN, type=REVERSE}
Choose a card if you have a playable one:
1- Card{color=BLUE, type=NUMBER, number=1}
2- Card{color=BLUE, type=NUMBER, number=4}
3- Card{color=YELLOW, type=NUMBER, number=2}
4- Card{color=BLUE, type=NUMBER, number=5}
5- Card{color=WILD, type=WILD}
6- Card{color=RED, type=NUMBER, number=1}
Enter the card index that you would like to play:
5
This is WILD card, you have to choose a color:
1- RED
2- BLUE
3- GREEN
4- YELLOW
Enter a color index:
2
```

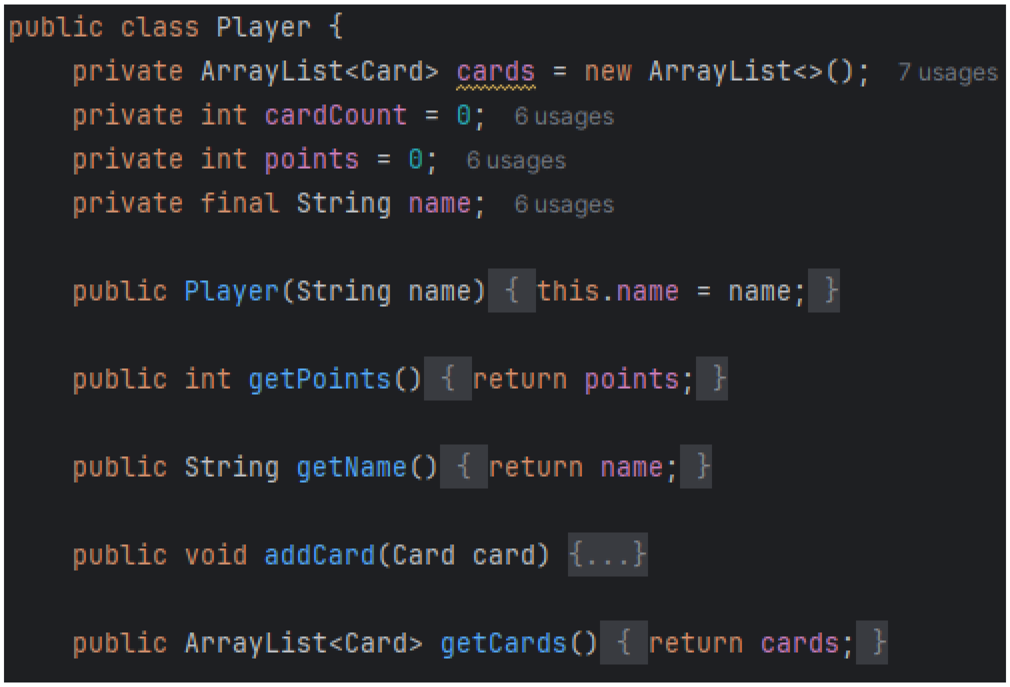

```
Khaled! it's your turn
This is the card on top:  Card{color=WILD, type=WILD}
Choose a card if you have a playable one:
1- Card{color=RED, type=NUMBER, number=1}
Enter the card index that you would like to play:
1
The round has finished!
Khaled won the round. They have 236 points.
```



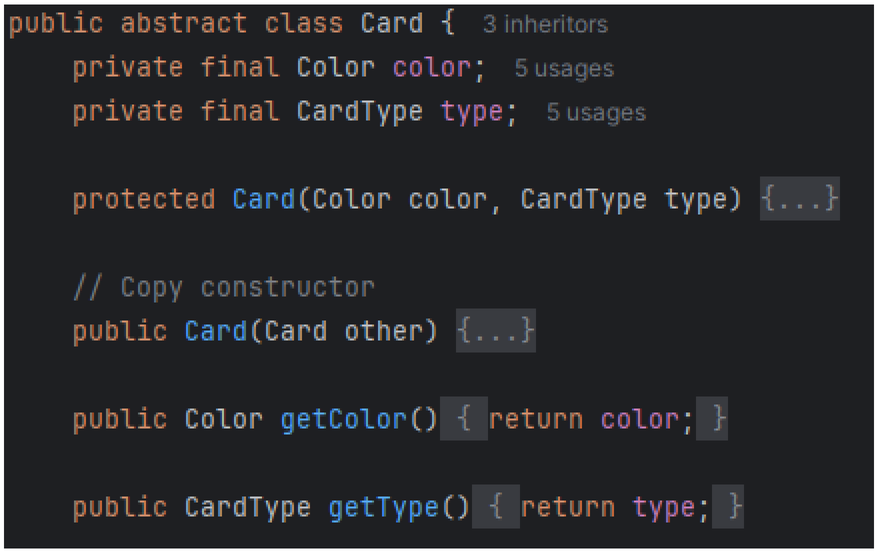
```
Ahmad! it's your turn
This is the card on top:  Card{color=YELLOW, type=NUMBER, number=6}
Choose a card if you have a playable one:
1- Card{color=WILD, type=WILD_DRAW_FOUR}
2- Card{color=YELLOW, type=DRAW_TWO}
3- Card{color=YELLOW, type=NUMBER, number=0}
4- Card{color=RED, type=NUMBER, number=4}
5- Card{color=BLUE, type=NUMBER, number=2}
6- Card{color=BLUE, type=DRAW_TWO}
7- Card{color=RED, type=NUMBER, number=2}
Enter the card index that you would like to play:
2
Player Mosaab draws 2 cards
```

Object-Oriented Design

- **Encapsulation**: I was keen on making the data encapsulated carfully in its class, I used appropriate access modifiers for the fields methods and used `final` when needed, I put a suitable getters and setters for fields that need to be processed for operations outside its class.

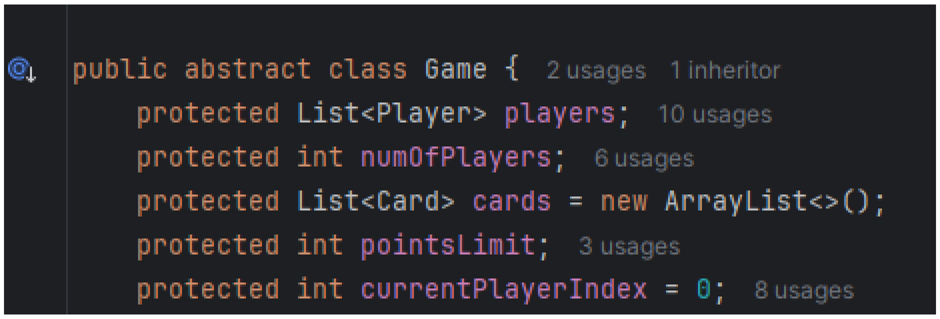


```
public class Player {
    private ArrayList<Card> cards = new ArrayList<>();  7 usages
    private int cardCount = 0;  6 usages
    private int points = 0;  6 usages
    private final String name;  6 usages

    public Player(String name) { this.name = name; }

    public int getPoints() { return points; }

    public String getName() { return name; }

    public void addCard(Card card) {...}

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

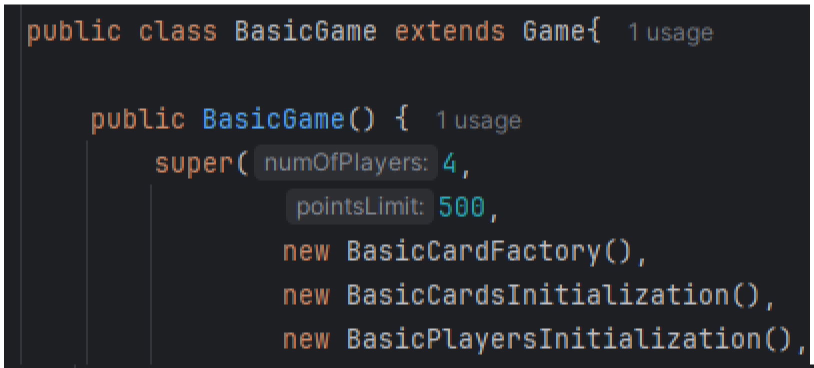


```java
public abstract class Card {  3 inheritors
    private final Color color;  5 usages
    private final CardType type;  5 usages

    protected Card(Color color, CardType type) {...}

    // Copy constructor
    public Card(Card other) {...}

    public Color getColor() { return color; }

    public CardType getType() { return type; }
```

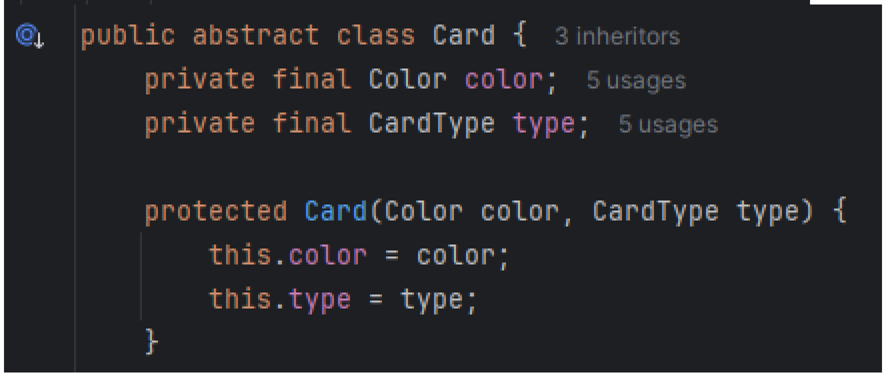
- **Inheritance**: Inheritance played a critical rule in the project design, especially in enabling other developers who use the game engine to extend the interfaces and implement their own games. The main point in the program was the abstract `Game` class that should be implemented by a concrete game (`BasicGame` for example), and with different **strategies** and their implementations, this way we can impose the developer to implement or override certain method to build a game.



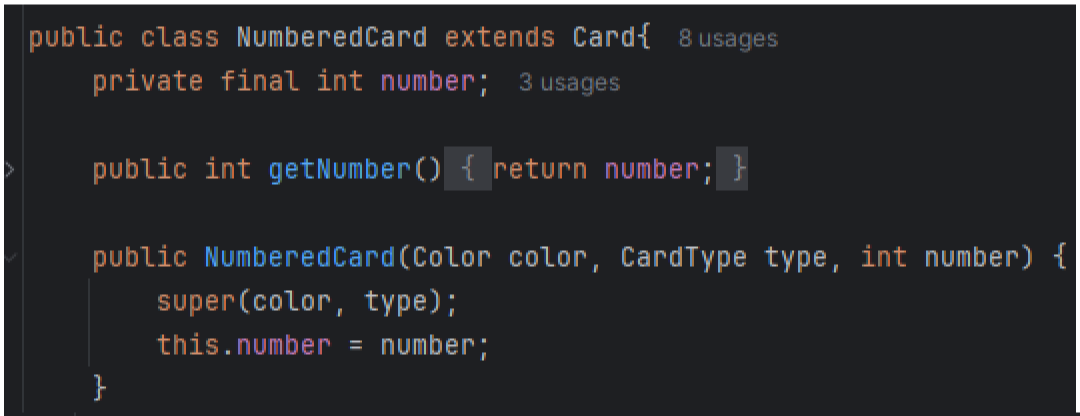
```java
public abstract class Game {  2 usages  1 inheritor
    protected List<Player> players;  10 usages
    protected int numOfPlayers;  6 usages
    protected List<Card> cards = new ArrayList<>();
    protected int pointsLimit;  3 usages
    protected int currentPlayerIndex = 0;  8 usages
```



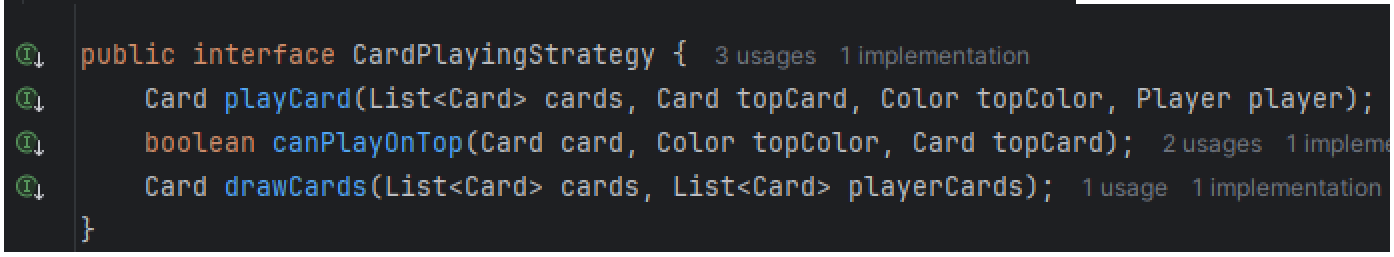
```java
public class BasicGame extends Game{  1 usage

    public BasicGame() {  1 usage
        super( numOfPlayers: 4,
                pointsLimit: 500,
                new BasicCardFactory(),
                new BasicCardsInitialization(),
                new BasicPlayersInitialization(),
```



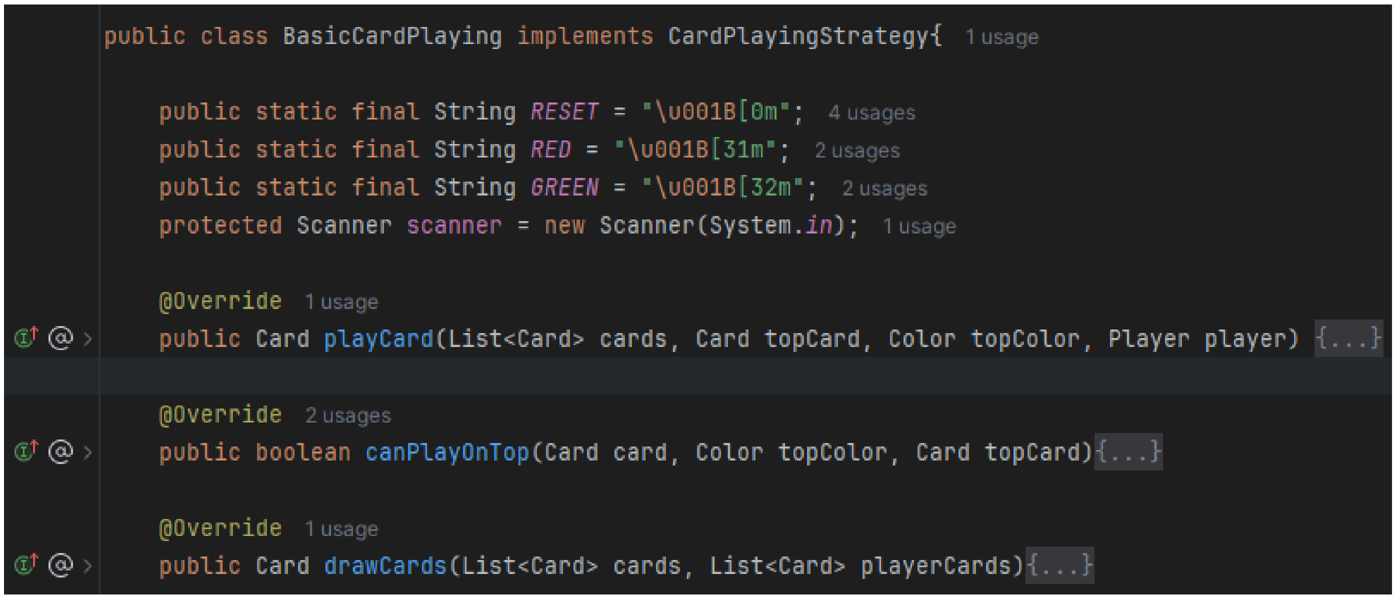
```java
public abstract class Card {  3 inheritors
    private final Color color;  5 usages
    private final CardType type;  5 usages

    protected Card(Color color, CardType type) {
        this.color = color;
        this.type = type;
    }
```



```
public class NumberedCard extends Card{  8 usages
    private final int number;  3 usages

    public int getNumber() { return number; }

    public NumberedCard(Color color, CardType type, int number) {
        super(color, type);
        this.number = number;
    }
```

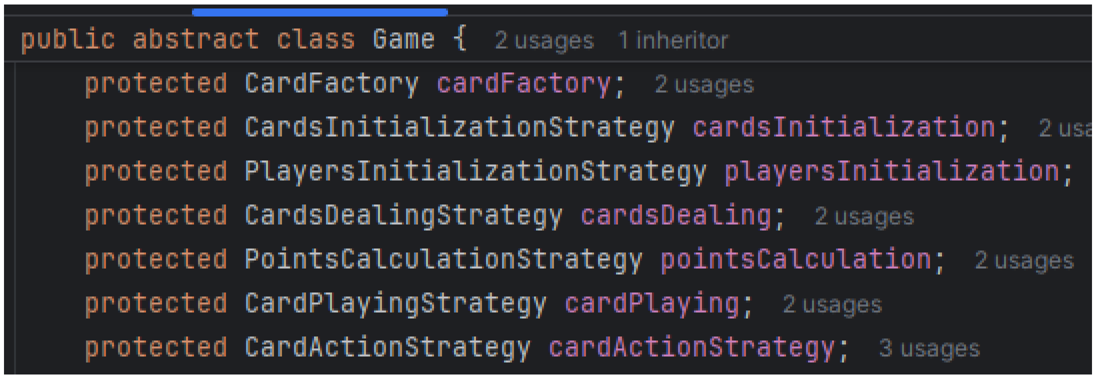


```
public interface CardPlayingStrategy {  3 usages  1 implementation
    Card playCard(List<Card> cards, Card topCard, Color topColor, Player player);
    boolean canPlayOnTop(Card card, Color topColor, Card topCard);  2 usages  1 impleme
    Card drawCards(List<Card> cards, List<Card> playerCards);  1 usage  1 implementation
}
```

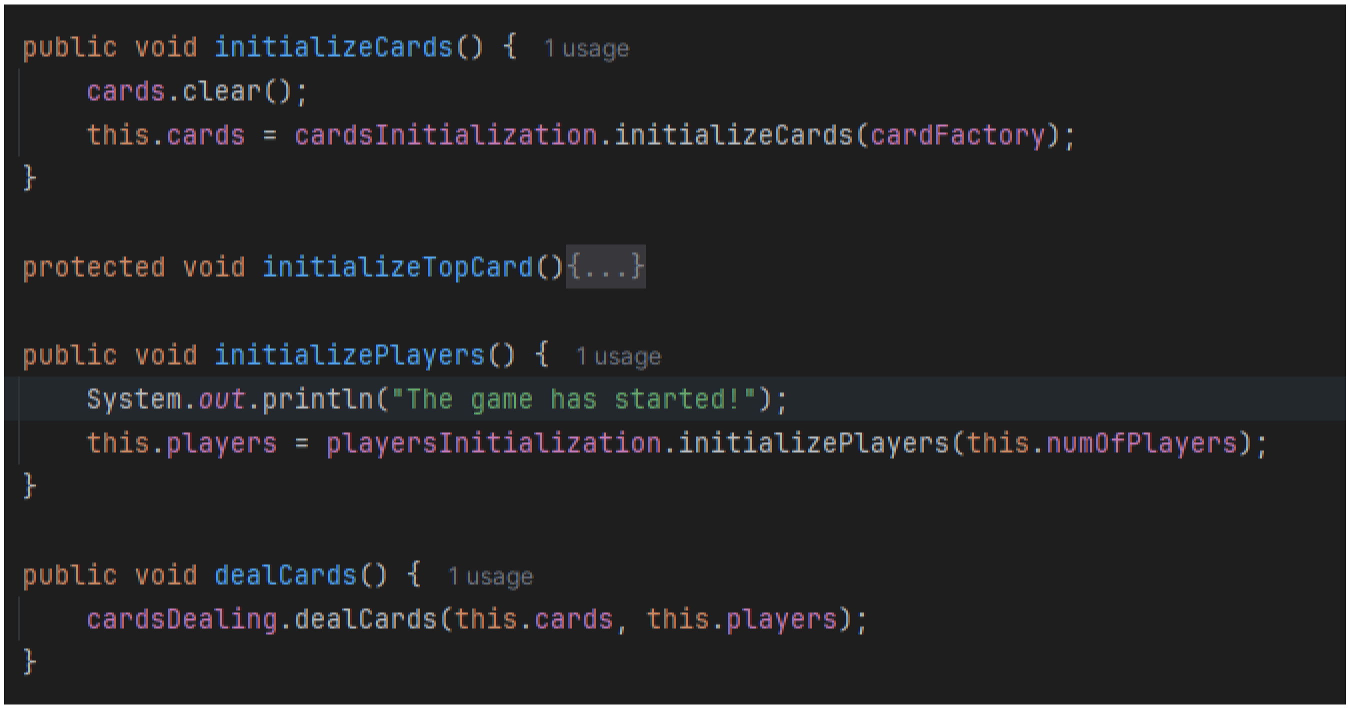


```
public class BasicCardPlaying implements CardPlayingStrategy{  1 usage

    public static final String RESET = "\u001B[0m";  4 usages
    public static final String RED = "\u001B[31m";  2 usages
    public static final String GREEN = "\u001B[32m";  2 usages
    protected Scanner scanner = new Scanner(System.in);  1 usage

    @Override  1 usage
    public Card playCard(List<Card> cards, Card topCard, Color topColor, Player player) {...}

    @Override  2 usages
    public boolean canPlayOnTop(Card card, Color topColor, Card topCard){...}

    @Override  1 usage
    public Card drawCards(List<Card> cards, List<Card> playerCards){...}
```

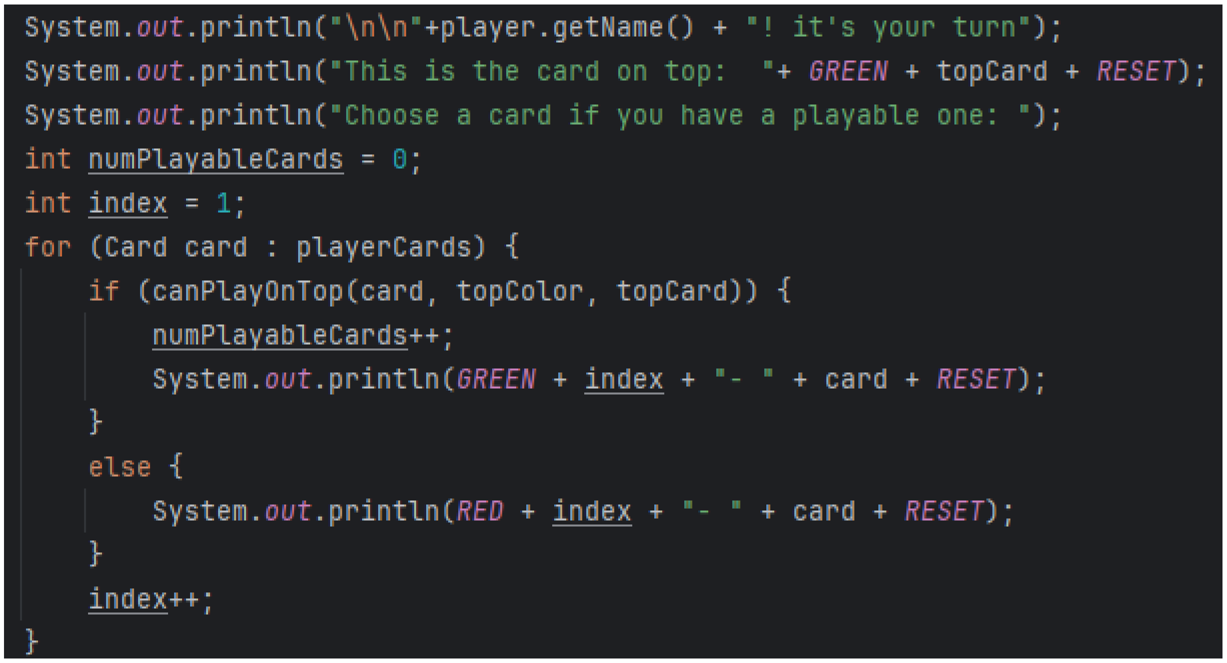
- **Polymorphism**: I used the concept of polymorphism many times in my implementation, especially with `Card` abstract class and its concrete subclasses (`NumberedCard` , `ActionCard` , and `WildCard`). I also used the polymorphism when strategies manipulation in `Game` class, so I used a references of interfaces types, and created a concrete subclass in the runtime.



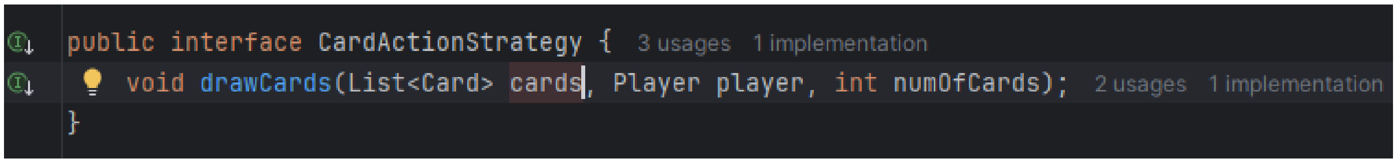
```
public abstract class Game {  2 usages  1 inheritor
    protected CardFactory cardFactory;  2 usages
    protected CardsInitializationStrategy cardsInitialization;  2 usa
    protected PlayersInitializationStrategy playersInitialization;
    protected CardsDealingStrategy cardsDealing;  2 usages
    protected PointsCalculationStrategy pointsCalculation;  2 usages
    protected CardPlayingStrategy cardPlaying;  2 usages
    protected CardActionStrategy cardActionStrategy;  3 usages
```

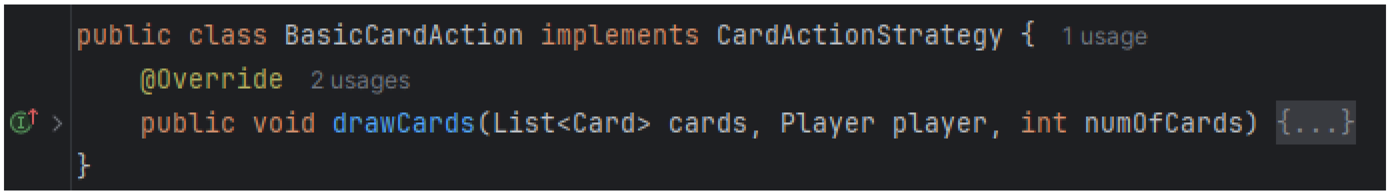

```
public void initializeCards() {  1 usage
    cards.clear();
    this.cards = cardsInitialization.initializeCards(cardFactory);
}

protected void initializeTopCard(){...}

public void initializePlayers() {  1 usage
    System.out.println("The game has started!");
    this.players = playersInitialization.initializePlayers(this.numOfPlayers);
}

public void dealCards() {  1 usage
    cardsDealing.dealCards(this.cards, this.players);
}
```

The polymorphism is also used with **cards**:

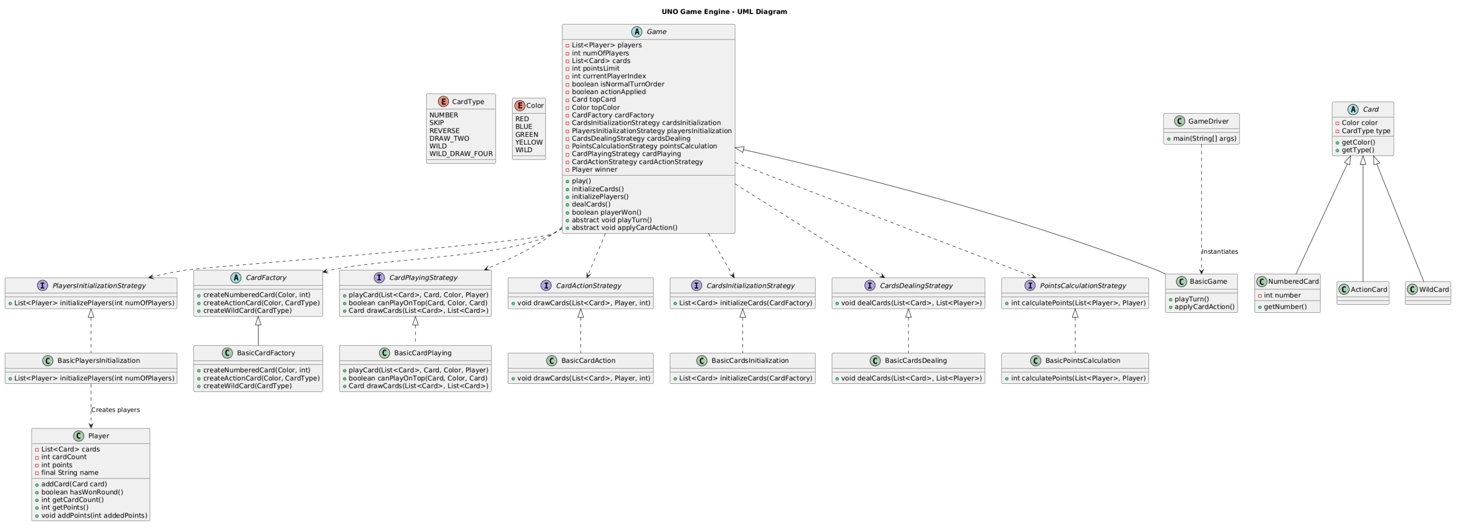

```
System.out.println("\n\n"+player.getName() + "! it's your turn");
System.out.println("This is the card on top:  "+ GREEN + topCard + RESET);
System.out.println("Choose a card if you have a playable one: ");
int numPlayableCards = 0;
int index = 1;
for (Card card : playerCards) {
    if (canPlayOnTop(card, topColor, topCard)) {
        numPlayableCards++;
        System.out.println(GREEN + index + "- " + card + RESET);
    }
    else {
        System.out.println(RED + index + "- " + card + RESET);
    }
    index++;
}
```

- **Abstraction**: I used abstraction in my design with both **interfaces** and **abstract classes**, like in `Card` hierarchy, `Game` abstract class, and all **strategies** that are used by `Game` .


```
public interface CardActionStrategy {  3 usages  1 implementation
    void drawCards(List<Card> cards, Player player, int numOfCards);  2 usages  1 implementation
}
```


```
public class BasicCardAction implements CardActionStrategy {  1 usage
    @Override  2 usages
    public void drawCards(List<Card> cards, Player player, int numOfCards) {...}
}
```

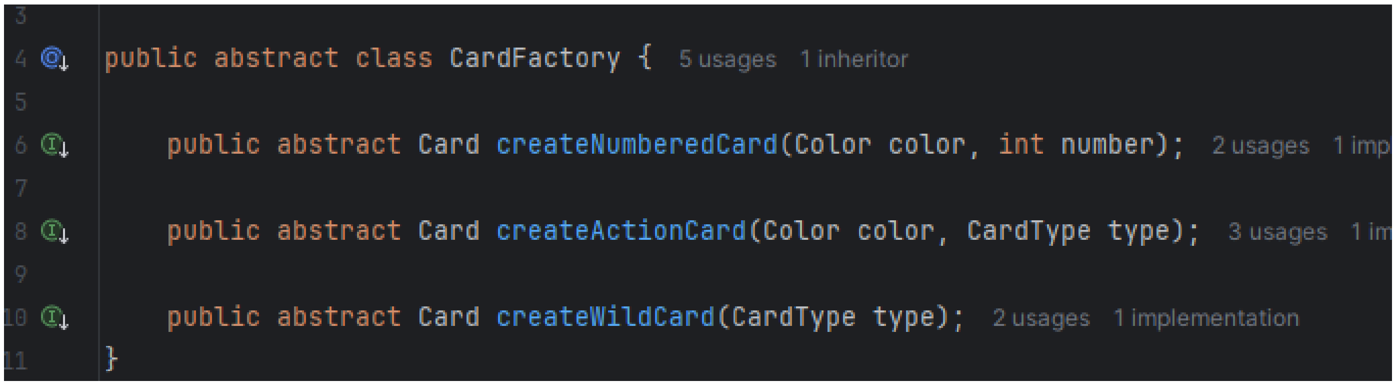
This is the detailed **UML** diagram of the system, showing the fields, methods and the relationships between interfaces and classes in a clear way:


UNO Game Engine - UML Diagram
CardType
NUMBER
SKIP
REVERSE
DRAW_TWO
WILD
WILD_DRAW_FOUR
Color
RED
BLUE
GREEN
YELLOW
WILD
Game
List<Player> players
int numOfPlayers
List<Card> cards
int pointsLimit
int currentPlayerIndex
boolean isNormalTurnOrder
boolean actionApplied
Card topCard
Color topColor
CardFactory cardFactory
CardsInitializationStrategy cardsInitialization
PlayersInitializationStrategy playersInitialization
CardsDealingStrategy cardsDealing
PointsCalculationStrategy pointsCalculation
CardPlayingStrategy cardPlaying
CardActionStrategy cardActionStrategy
Player winner
play()
initializeCards()
initializePlayers()
dealCards()
boolean playerWon()
abstract void playTurn()
abstract void applyCardAction()
GameDriver
main(String[] args)
Instantiates
Card
Color color
CardType type
getColor()
getType()
PlayersInitializationStrategy
List<Player> initializePlayers(int numOfPlayers)
CardFactory
createNumberedCard(Color, int)
createActionCard(Color, CardType)
createWildCard(CardType)
CardPlayingStrategy
playCard(List<Card>, Card, Color, Player)
boolean canPlayOnTop(Card, Color, Card)
Card drawCards(List<Card>, List<Card>)
CardActionStrategy
void drawCards(List<Card>, Player, int)
CardsInitializationStrategy
List<Card> initializeCards(CardFactory)
CardsDealingStrategy
void dealCards(List<Card>, List<Player>)
PointsCalculationStrategy
int calculatePoints(List<Player>, Player)
BasicGame
playTurn()
applyCardAction()
NumberedCard
int number
getNumber()
ActionCard
WildCard
BasicPlayersInitialization
List<Player> initializePlayers(int numOfPlayers)
BasicCardFactory
createNumberedCard(Color, int)
createActionCard(Color, CardType)
createWildCard(CardType)
BasicCardPlaying
playCard(List<Card>, Card, Color, Player)
boolean canPlayOnTop(Card, Color, Card)
Card drawCards(List<Card>, List<Card>)
BasicCardAction
void drawCards(List<Card>, Player, int)
BasicCardsInitialization
List<Card> initializeCards(CardFactory)
BasicCardsDealing
void dealCards(List<Card>, List<Player>)
BasicPointsCalculation
int calculatePoints(List<Player>, Player)
Creates players
Player
List<Card> cards
int cardCount
int points
final String name
addCard(Card card)
boolean hasWonRound()
int getCardCount()
int getPoints()
void addPoints(int addedPoints)

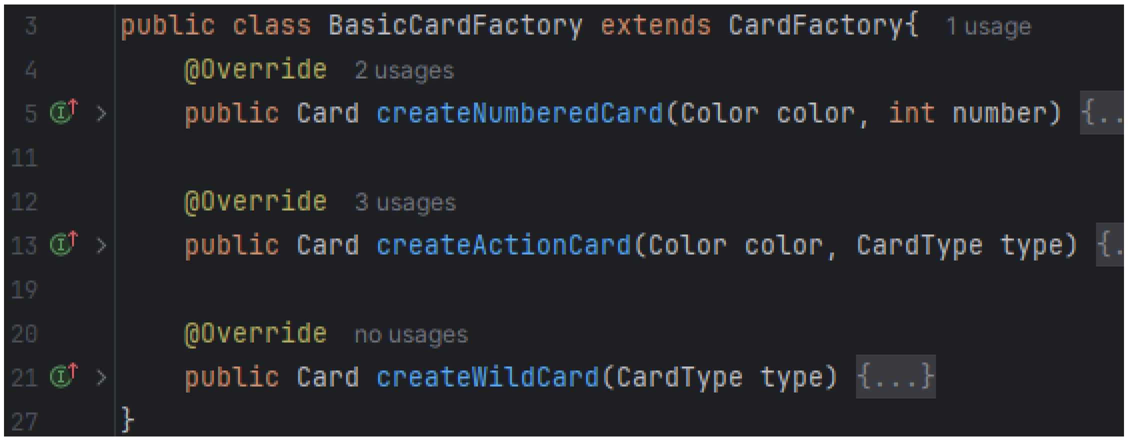
Design Patterns

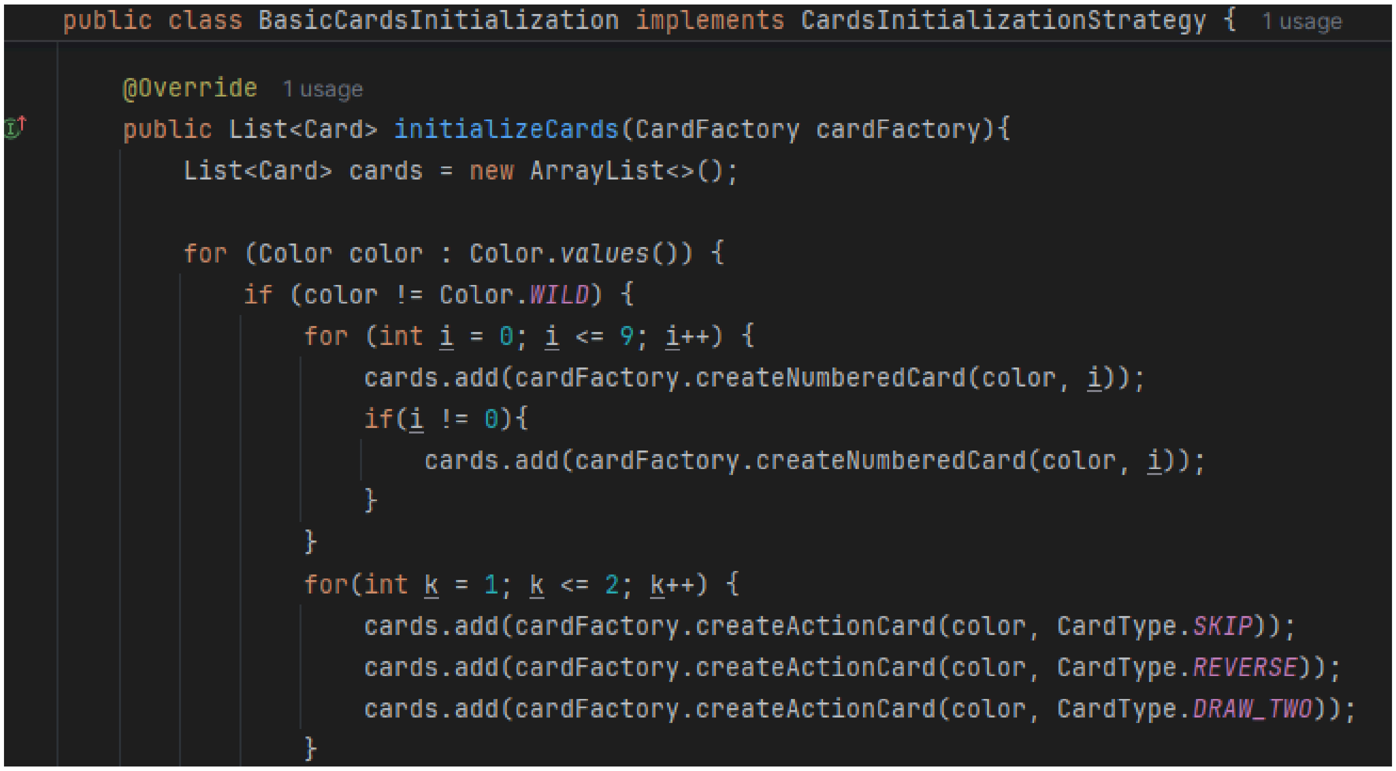
The most challenging aspect about this assignment was the implementation of different design patterns, it was my first time implementing this topic in a real application:

- **Factory Pattern**: I used Factory pattern to create the set of cards in the game, I created an abstract class called `CardFactory` and enabled other developer to implement their own concrete classes to create Uno cards as they want, with the number they want (they can't change the `CardType` or `CardColor`).
  An instance of that concrete factory will be one of the fields in the `Game` class, it will be used every round to initialize cards as wanted in the variation rules.

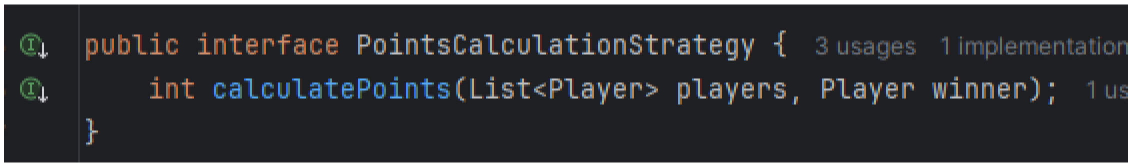


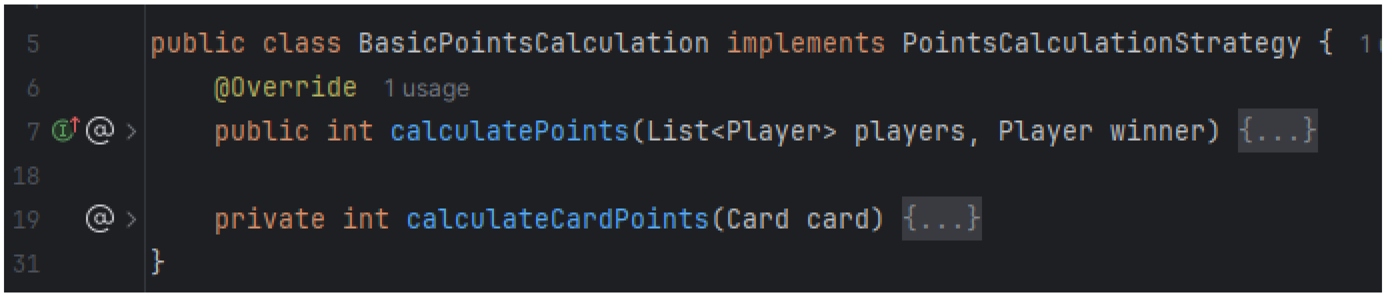
```
public abstract class CardFactory {  5 usages  1 inheritor

    public abstract Card createNumberedCard(Color color, int number);  2 usages  1 imp

    public abstract Card createActionCard(Color color, CardType type);  3 usages  1 im

    public abstract Card createWildCard(CardType type);  2 usages  1 implementation
}
```


```
public class BasicCardFactory extends CardFactory{  1 usage
    @Override  2 usages
    public Card createNumberedCard(Color color, int number) {..

    @Override  3 usages
    public Card createActionCard(Color color, CardType type) {.

    @Override  no usages
    public Card createWildCard(CardType type) {...}
}
```

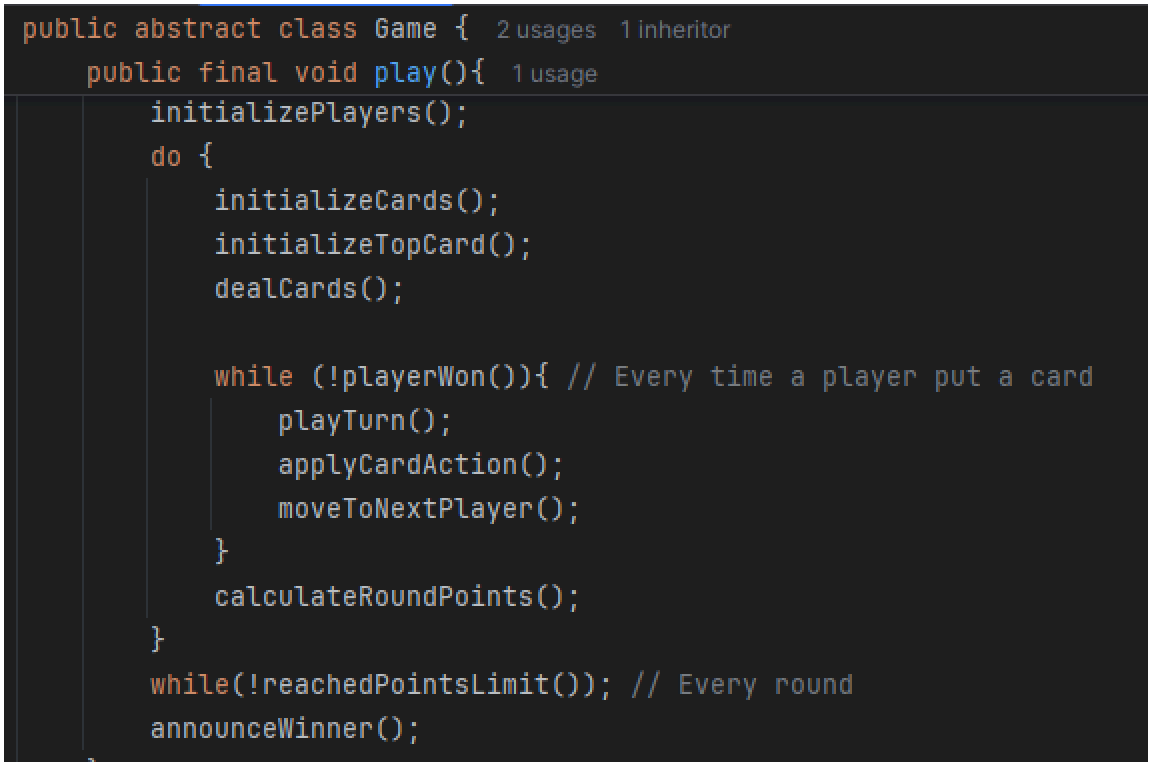

```
public class BasicCardsInitialization implements CardsInitializationStrategy {  1 usage

    @Override  1 usage
    public List<Card> initializeCards(CardFactory cardFactory){
        List<Card> cards = new ArrayList<>();

        for (Color color : Color.values()) {
            if (color != Color.WILD) {
                for (int i = 0; i <= 9; i++) {
                    cards.add(cardFactory.createNumberedCard(color, i));
                    if(i != 0){
                        cards.add(cardFactory.createNumberedCard(color, i));
                    }
                }
                for(int k = 1; k <= 2; k++) {
                    cards.add(cardFactory.createActionCard(color, CardType.SKIP));
                    cards.add(cardFactory.createActionCard(color, CardType.REVERSE));
                    cards.add(cardFactory.createActionCard(color, CardType.DRAW_TWO));
                }
```

- **Strategy Pattern**: Using strategy pattern was one of the most critical points in the system, this ability of decoupling the algorithms form the `Game` class was great, it gives the ability to use a set of different selected rules in one concrete class without modifying any previous code.


```
public interface PointsCalculationStrategy {  3 usages  1 implementation
    int calculatePoints(List<Player> players, Player winner);  1 us
}
```


```
public class BasicPointsCalculation implements PointsCalculationStrategy {  1
    @Override  1 usage
    public int calculatePoints(List<Player> players, Player winner) {...}

    private int calculateCardPoints(Card card) {...}
}
```

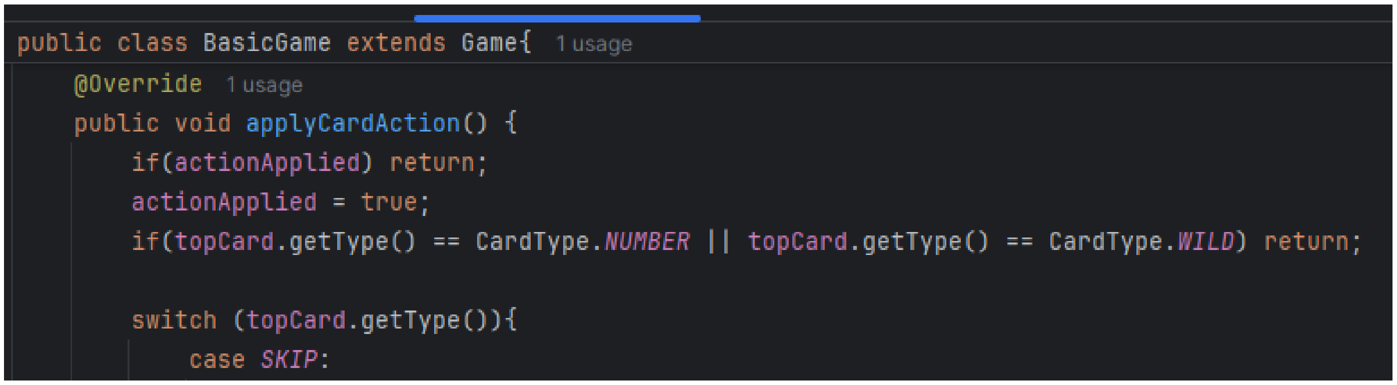
- **Template Method Pattern**: I created the method `play()` in `Game` to be a template method, it represents the main skeleton for the execution of other method, I made it `final` so subclasses cannot override it to force them to follow its steps.



```java
public abstract class Game {  2 usages  1 inheritor
    public final void play(){  1 usage
        initializePlayers();
        do {
            initializeCards();
            initializeTopCard();
            dealCards();

            while (!playerWon()){ // Every time a player put a card
                playTurn();
                applyCardAction();
                moveToNextPlayer();
            }
            calculateRoundPoints();
        }
        while(!reachedPointsLimit()); // Every round
        announceWinner();
```

Clean Code Principles

- Meaningful names: I was careful with naming classes, methods, and fields, even local variables, I made sure that all names are **clear**, **specific** and **descriptive** for their functionalities.
- Factory methods: I used a factory method to create all types of cards instead of construcotrs.
- Methods: Methods arguments was reasonable, and I tried to keep them simple and fail fast as possible to reduce complexity.



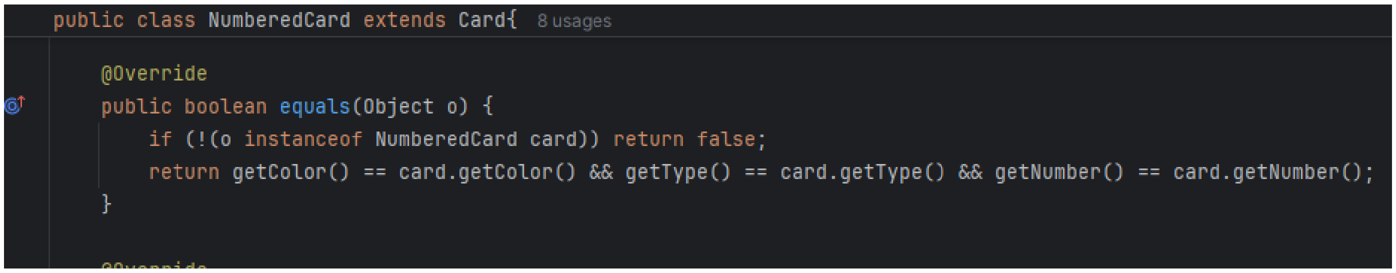
```java
public class BasicGame extends Game{  1 usage
    @Override  1 usage
    public void applyCardAction() {
        if(actionApplied) return;
        actionApplied = true;
        if(topCard.getType() == CardType.NUMBER || topCard.getType() == CardType.WILD) return;

        switch (topCard.getType()){
            case SKIP:
```

- Avoid complex nested ternary expressions.
- Class should have only one reason to change (**SRP**): I divided classes functionalities to apply the single responsibility principle.

- Coupling: I kept a healthy degree of dependency between classes.
- Write your code to interfaces, not concretes: This was obvious in my design especially when I used Strategy pattern.

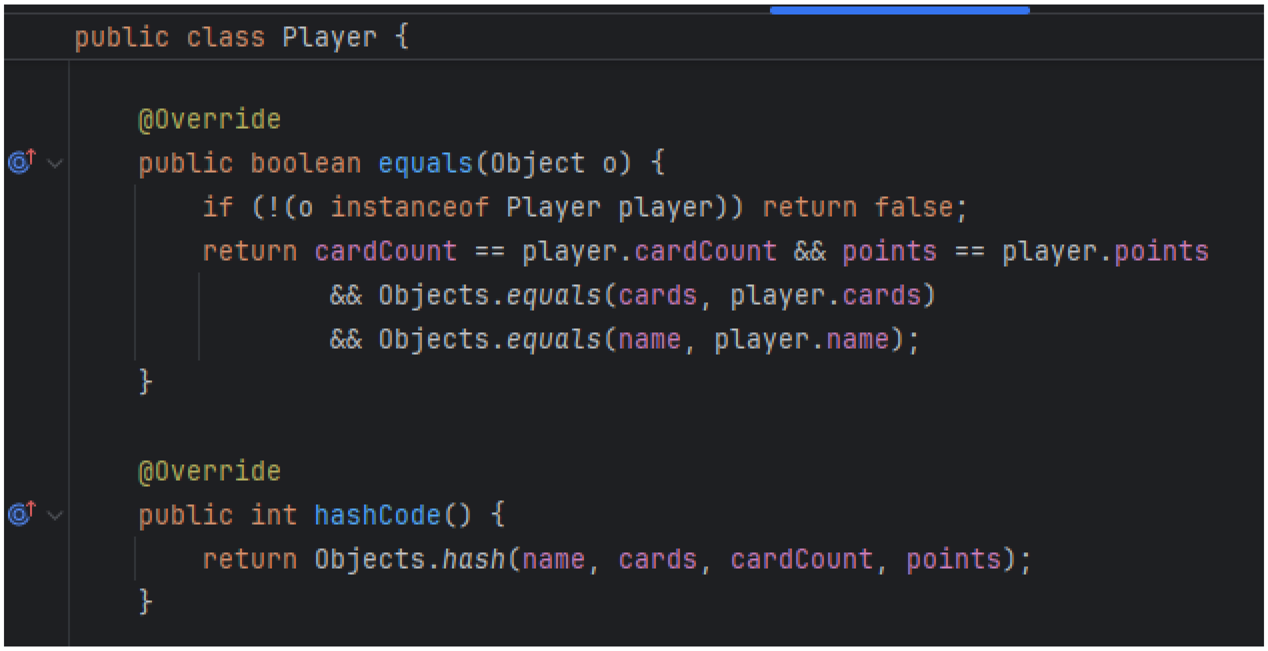
Effective Java

- Favoring composition over inheritance: I used that and the solutin was the strategy pattern with composition of strategies objects.
- Using appropriate collections and generics: I used `List` and `ArrayList` with cards and players, their built in methods was very helpful.
- Consider static factory methods instead of constructors: I did that with `CardFactory`.
- Prefer dependency injection to hardwiring resources: I used something similar with `BasicGame` class, I am not sure if it was a real dependency injection, but I think so.
- Avoid creating unnecessary objects: In general, I followed that principle, but I was creating a new set of cards every round instead of collecting cards again, I think I should have found a better solution.
- Eliminate obsolete object references: I did not used that ever.
- Avoid finalizers and cleaners: I did so.
- Obey the general contract when overriding `equals`.

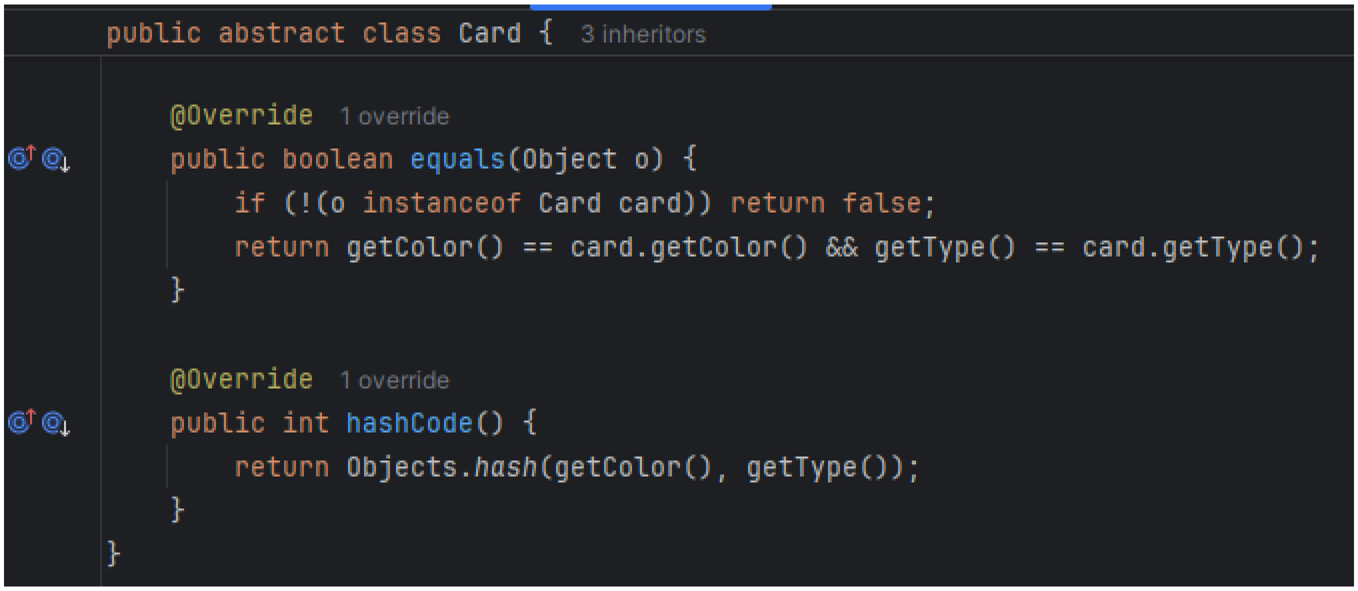


```java
public class NumberedCard extends Card{  8 usages

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof NumberedCard card)) return false;
        return getColor() == card.getColor() && getType() == card.getType() && getNumber() == card.getNumber();
    }
```

- Always override `hashCode` when you override `equals`.

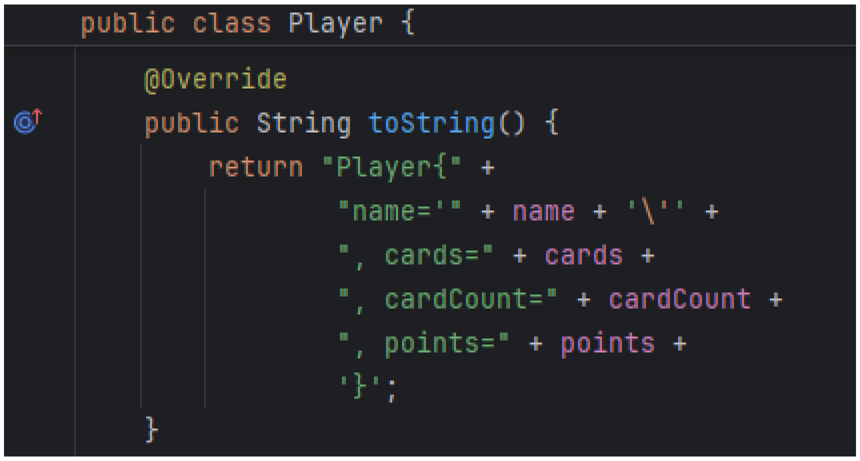


```java
public class Player {

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Player player)) return false;
        return cardCount == player.cardCount && points == player.points
                && Objects.equals(cards, player.cards)
                && Objects.equals(name, player.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, cards, cardCount, points);
    }
```

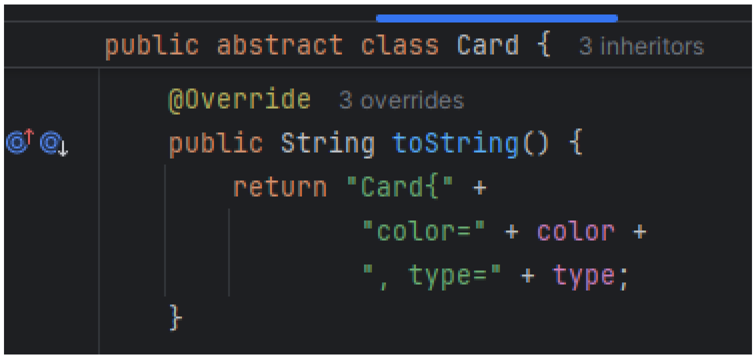


```java
public abstract class Card {  3 inheritors

    @Override  1 override
    public boolean equals(Object o) {
        if (!(o instanceof Card card)) return false;
        return getColor() == card.getColor() && getType() == card.getType();
    }

    @Override  1 override
    public int hashCode() {
        return Objects.hash(getColor(), getType());
    }
}
```

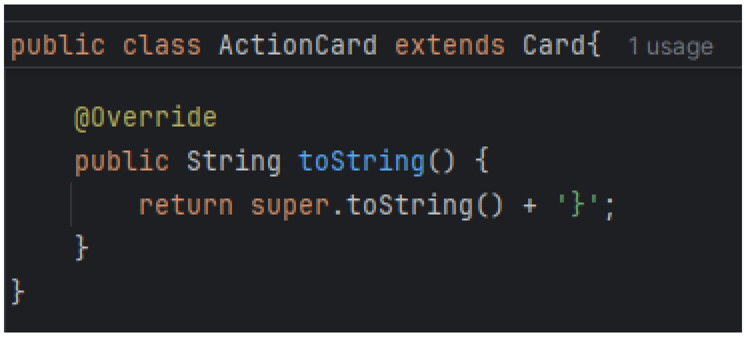
- Always override `toString`



```java
public class Player {
    @Override
    public String toString() {
        return "Player{" +
                "name='" + name + '\'' +
                ", cards=" + cards +
                ", cardCount=" + cardCount +
                ", points=" + points +
                '}';
    }
```



```
public abstract class Card {  3 inheritors
    @Override  3 overrides
    public String toString() {
        return "Card{" +
                "color=" + color +
                ", type=" + type;
    }
```



```
public class ActionCard extends Card{  1 usage

    @Override
    public String toString() {
        return super.toString() + '}';
    }
}
```

SOLID Principles

- **S**ingle Responsibility Principle (SRP): I wrote many strategies classes to ensure the single responsibility for every class.
- **O**pen-Closed Principle (OCP): My code is open for extention by other developers without the need for modifying any existing code.
- **L**iskov Substitution Principle (LSP): This was obvious with `Card` class and its subclasses, every class was interchangable with the superclass as Liskov Principle (with playing strategies also).
- **I**nterface Segregation Principle (ISP).
- **D**ependency Inversion Principle (DIP): All classes in the system was dependent on the abstraction, this gives the system the ability to adapt different variations.

Conclusion

This was a great experience, I tried many new things and was forced to read more about different important topics, I used design patterns effectively and solved all the problems and bugs to gain the knowledge and experience by practice.