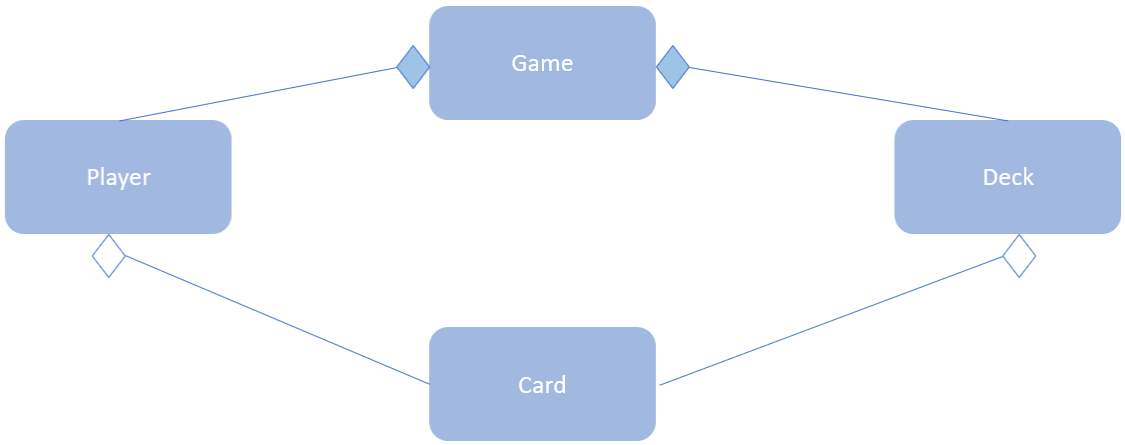
Uno Engine (Ungine)

Uno is a card game containing colorful cards with specific rules that are different from the rules of other card games. I was asked to create an Uno game engine that allows other developers to create their own variations of the Uno game. The developer can create his own rules, cards or extend the code easily to satisfy his needs. In this report, I will discuss the implementation of my Uno game engine. The implementation highly takes into consideration the S.O.L.I.D principles and utilizes several design patterns such as strategy pattern, mediator pattern, singleton pattern and decorator pattern. I am going to discuss the following topics respectively: **OOP design, Main classes, Defending against SOLID principles, Design patterns, Built-in features, and Future work.**

OOP design



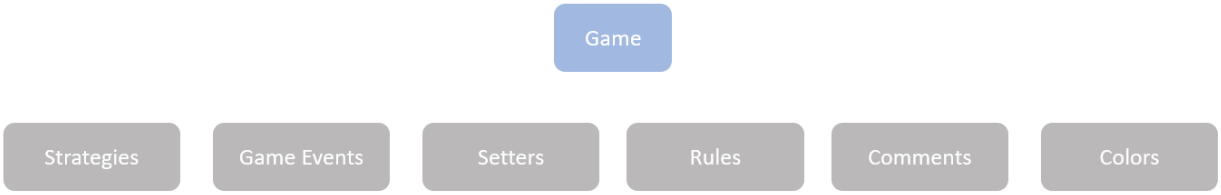
In my design, the Game class acts as a mediator between the players themselves, and between the players and the deck. When I designed the relations between the main classes above, I tried to minimize the direct relations as much as possible. But one line that could not be removed is the one between the Player and the Card classes. Yes, the player will communicate with the Deck class through the Game class to get his cards, but those cards will be under his responsibility

now. So, I had to make a relation between the Player and the Card too since the player holds a list of cards. The relationship between the Game class and the Player and the Deck classes is composition, since they are created inside the Game class itself, if the game instance is destroyed, they are destroyed too. While the relationship between The Card class and the Deck and the Player classes is aggregation. Yes, the Deck class creates the cards and holds the references of them, so normally it would be if the deck instance is destroyed then the cards must be destroyed too. But we must consider the cards held by the players. We conclude that if the deck instance is destroyed, then the cards are not necessarily destroyed, since the players might hold some of them.

Main classes

1. **Game class:**

Now I will discuss the main part of my engine, the Game class. The mediator between the players and the deck. This class must be extended by another class to create your own variation of the game. The class is flexible in a way that allows you to add new rules to the game, whatever you think about, by utilizing the idea of player comments. The player is prompted to add any comments at any chosen time in the game (chosen by the developer). These comments are processed usually to satisfy specific rules, either an existing rule that comes with the engine, or a new added one by the developer. One example could be the rule of forcing the player to say Uno when he only has a single card. The player is always prompted to add a comment after he plays a card in case he wants to comment “Uno”. Then the rule named “SayUnoRule” will check if the player has already said Uno when he only had a single card. So, this is how you can add any customization to the game, next we will discuss the Game class in detail.



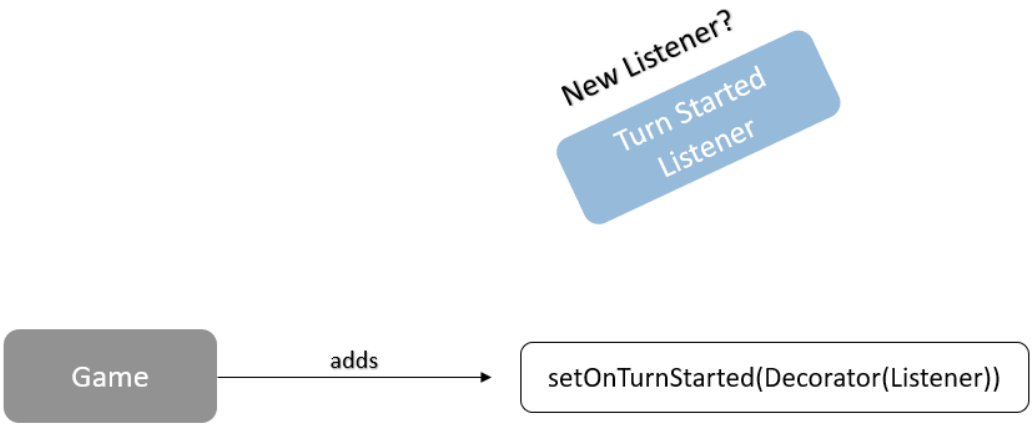
The class contains several aspects, considered as tools, to allow you to further customize your variation. Let's discuss each one in details:

Strategies:

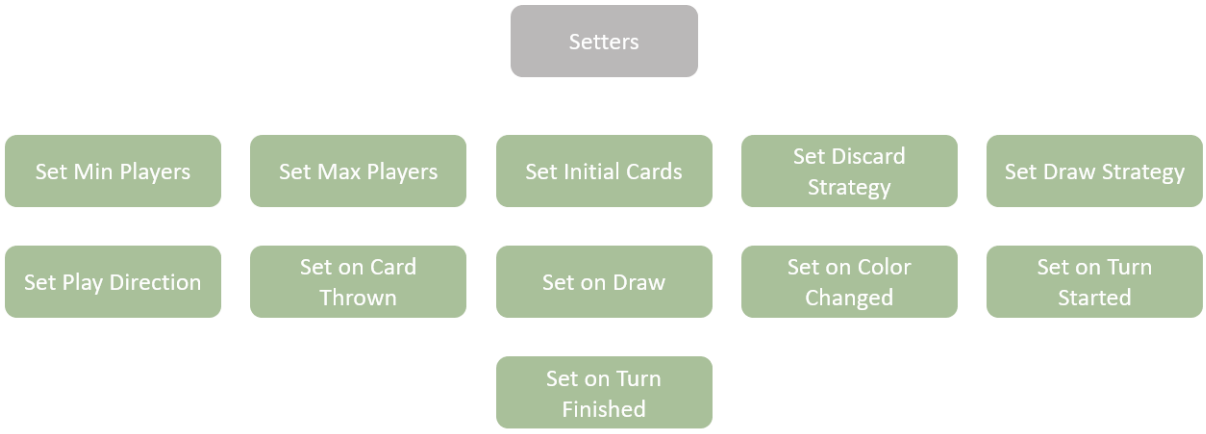
- Card discard strategy: responsible for deciding how the player discards cards in normal conditions, should he discard a single card, or multiple ones.
- Draw strategy: responsible for deciding how the player draws cards in normal conditions, should he draw a single card, or multiple ones.

Game Events:

The events idea allows the developer to be notified in code on important events in the game, such as the ones in the figure above. This feature is implemented using a combination of observer pattern and decorator pattern. Instead of using bare observer pattern I decided to combine it with the other one to avoid creating extra methods for adding and removing observers for each event. The developer can set the game event using its setter method. And by utilizing the GameEventDecorator class, the developer can create multiple listeners and determine their order of execution. The figure below shows how the developer can bind several listeners to a specific game event.

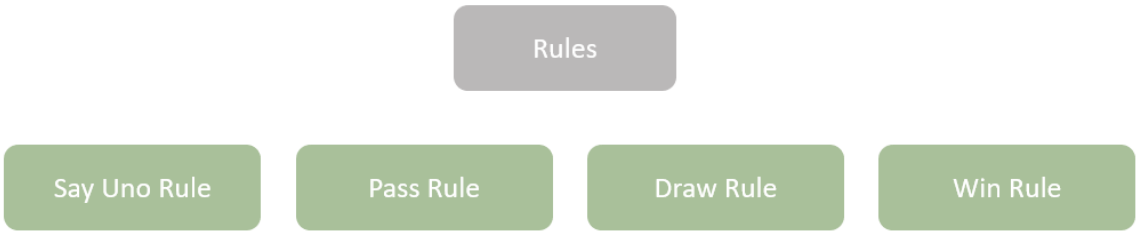


Setters:



As we can see, the developer has several options to customize the built in rules of the game to satisfy his own variation.

Rules:

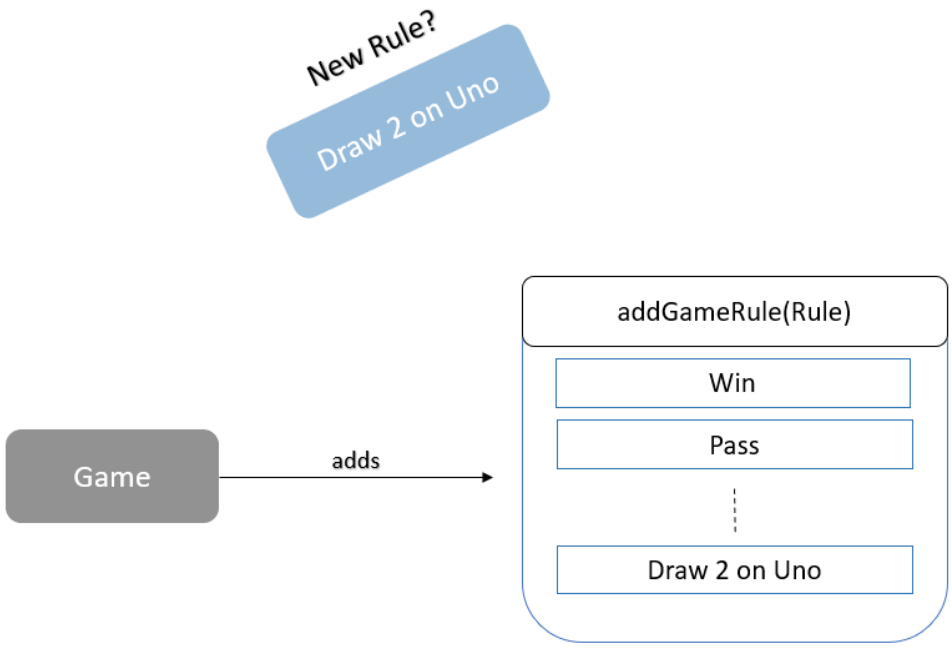


As we can see, the rules in the figure above are the built-in rules of the game.

- Say Uno Rule: checks whether the player has said Uno when he only had a single card.
- Pass Rule: checks whether the player has said pass. If so, it skips his turn.
- Draw Rule: checks whether the player has said draw. If so, it draws him/her a card depending on the draw strategy specified in the Game class.

- Win Rule: checks whether the player who played the last card has zero cards left. If so, he/she wins.

Each rule has a validator and a handler. It uses the validator to check whether the rule is satisfied at the moment of checking. And it uses the handler to apply the consequences of the rule. The developer can easily add a new rule by adding it to the Game object at runtime. Then the only thing the developer must do is to validate and handle the rule whenever he/she wants. The figure below demonstrates the process of adding a new rule. We could add a rule that forces the player to draw two cards instead of one if he wants to draw a card while he only has one.

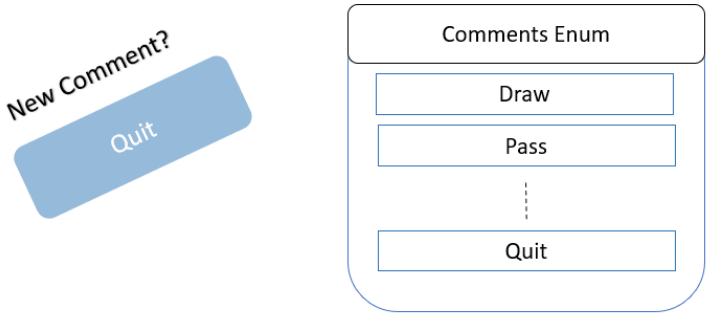


Comments:

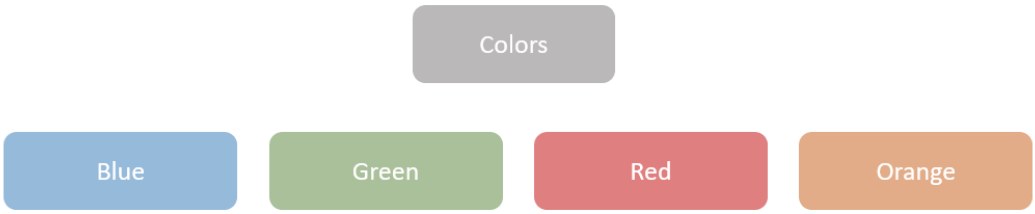
As we can see, the comments in the figure above are the built-in comments that the player can say.

- Uno: the player is prompted to say uno every time he has a single card.
- Pass: the player is prompted to say pass every time he has a turn.
- Draw: the player is prompted to say pass every time he has a turn.

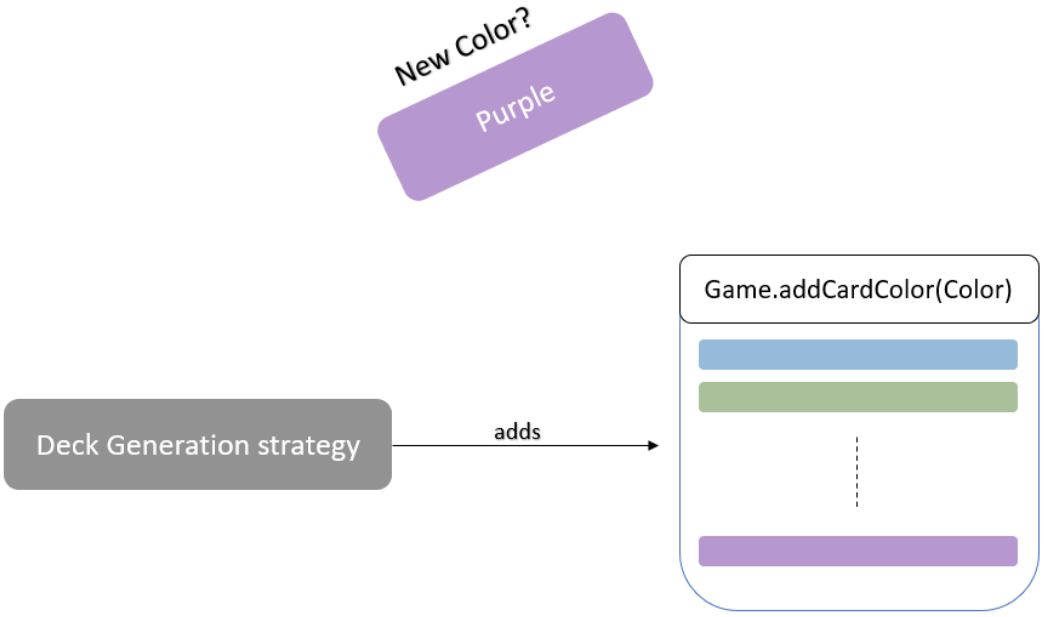
The player is prompted to add any comment at any instance of time the developer chooses. The developer can easily add a new available comment as clarified in the figure below.



Colors:

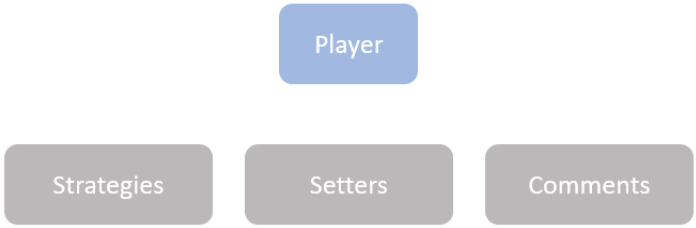


As we can see, the colors in the figure above are the available card colors in the game. The developer must add all the used colors to the Game class, this step must be done so when the player is prompted to choose a color, all available colors are displayed as choices. The figure below clarifies how to add a new color.



2. **Player class:**

Now let's move to the next class, the class allows customization through three aspects clarified in the figure below.



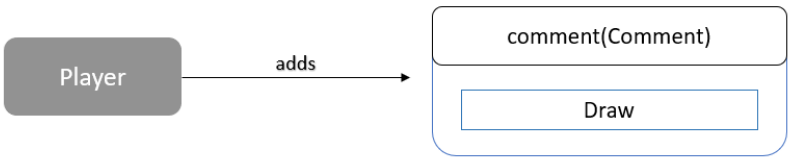
Strategies:

- Play turn strategy: decides the mechanism of playing the player's turn. It normally includes the prompting process for comments, allowing the player to discard a card and similar operations.
- Numbered question strategy: allows the developer to choose different ways to answer a multiple-choice question. This strategy decouples the relation between the real player and the bot player since a bot player will not answer the multiple-choice question in an interactive way.

Setters:

The Player class has setter methods for the strategies discussed above.

Comments:



The player can comment while playing his turn. In the example above, the player comments to draw a card from the deck. Then a corresponding rule checks if the player has commented to draw a card. If so, it gives him one or more cards depending on the draw strategy in the Game class.

3. **Deck class:**

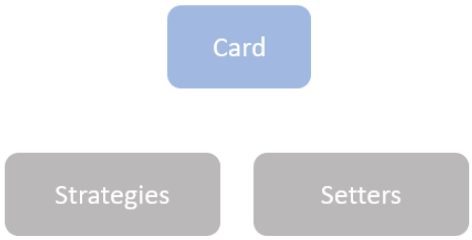
The next class is the Deck class, this one allows customization only through strategies.

Strategies:

The only built-in strategy is the one that defines how the deck is generated. This strategy must decide what colors are used for cards and add them to the Game class. Then it must create the cards included in the deck.

4. **Card class:**

The Card class can be customized through the following.



Strategies:

- Card action strategy: defines the result of playing a card. A numbered card will not do anything for instance, while a wild card will prompt the user to change the color of the card. Another one could penalize the next player by drawing two cards.
- Card play-over strategy: defines over what cards can a specific card be played. A numbered card can be played over another card with the same number or color. While a wild card can be played over any card.

Setters:

The Card class has setter methods for the strategies discussed above. Also, for the color of the card and the value of it. The value of a card might be used to calculate the score of each player at the end of the game.

Defending against S.O.L.I.D principles

1. **Single responsibility**

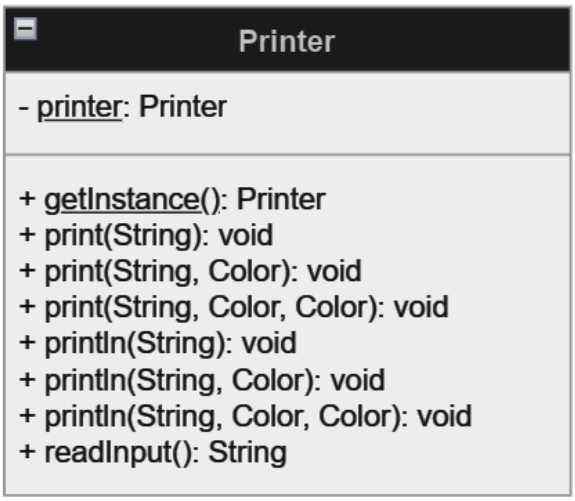
The game class is only responsible of game management related responsibilities. It is responsible of the following:

- Allowing players to join the game.
- Setting basic rules such as minimum players and initial number of cards.
- Setting other rules such as the card drawing and discarding strategies.
- Adding advanced comment-driven rules such as saying Uno when the player only has a single card.
- Dealing cards to players.
- Asking the player to play his turn.
- Switching the turn to the next player.
- Since the game is the mediator between the players and the deck, it allows players to draw cards through it.

Also, the player class is responsible of the following:

- Choosing a card.
- Interacting with any questions.
- Adding comments.

One strong example that supports this principle, is creating the Printer class which is responsible for printing information on the console (or other UI variations). It is split from the player or game classes since it will violate the principle. However, this printer class violates the principle by being responsible of reading input from the user too. This class uses the singleton pattern.



I am also a little concerned about the CardActionStrategy that takes the game instance as a parameter inside the performAction method.

```
public interface CardActionStrategy {
    7 implementations
    void performAction(Game game);
}
```

This way might increase the coupling, but on the other hand the developer can get way more customizations to do on new cards.

2. **Open/Close**

Utilizing design patterns, mostly the strategy design pattern, I was able to achieve a very extendible implementation. You can substitute the implementation of any feature by implementing the interface that supports that feature. Also, the decorator pattern was utilized to create multiple listeners of a game event, more about that in the design patterns section.

The bottom line, this principle was mainly satisfied by **encapsulating what changes from what stays the same.**

3. **Liskov's substitution**

```
public void playTurn() { turnStrategy.playTurn(getGame(), player: this); }
```

For this principle, I just had to make sure that no subclass overrides a super method to do something weird. Although the example below is not very related to the principle, I would like to discuss it.

Before implementing the playTurn method this way in the Player class, it was an abstract method that has different implementations between real player and bot player. Then I thought, if the developer gets very creative, he might want to change the nature of player's turn at runtime. For example, at the beginning of the game all players can choose to pass or draw a card even when they don't have any cards that can be played. Then when any player gets in a situation where he only has a single card in his hand, the behavior will change for him, he can't pass if he can't play, but he must draw a card. Of course, this is one way to achieve it. But the nice thing about my implementation is that the developer can achieve the same discussed feature by creating a rule named "NoPassOnUno" for example, without substituting the whole playTurn strategy. Moreover, the developer might achieve the same purpose by not adding the "Pass" comment to the list of choices when the player has only a single card.

I hope you can see how many options the developers can go with to achieve the same purpose.

4. **Interface segregation**

All interfaces have a single responsibility in a way that helps the developer to avoid implementing an interface that contains some methods he doesn't need.

An example where I was douting if I violate this principle is the DeckGenerationStrategy. But after a second thought, all deck generation strategies must define colors for the cards they are going to use for the generated cards.

```
public interface DeckGenerationStrategy {
    1 usage   1 implementation
    List<Card> generate();
    2 usages   1 implementation
    void addUsedColors();
}
```

5. **Dependency inversion**

- joinGame(Player player)
- throwCardOnTable(Card card)
- giveCard(Card card)
- performAction(Game game)
- addCardToBottom(Card card)

The method examples above support this principle, by **programming to the interface instead of programming to the implementation**.

Desing patterns

1. **Strategy pattern**

Utilized to support the open/close principle, it also allows us to change the behavior of a specific feature at runtime. One example of a spot where I utilized the pattern is the action strategy of a card, you might want to change the nature of the card on a specific event in the game. Or the card drawing strategy, where you might add some extreme drawing rules at the final stages of a game.

2. **Mediator pattern**

The game class acts like the mediator between the players themselves, and between the players and the deck.

3. **Observer pattern**

A modified version of the pattern was used. As we can see below, the GameEvent interface allows us to create special events in the game (Observables), then we can assign a value to that event (Observers) to execute some code when the event occurs. One good question would be that this way of observing only allows us to assign a single listener (observer) to each event. Check the next pattern to see how it can be utilized to attach several listeners.

```
public interface GameEvent {
    1 usage
    void onExecuted(Game game);
}
```

4. **Decorator pattern**

The base listener can be of type GameEvent, then we can create several listeners of type GameEventDecorator and stack them together as explained below.

```java
GameEvent l1 = game -> System.out.println("Listener 1");
GameEventDecorator l2 = new GameEventDecorator(l1) {
    2 usages
    @Override
    public void onExecuted(Game game) {
        getListener().onExecuted(game);
        System.out.println("Listener 2");
    }
};
setOnDraw(l2);
```

5. **Singleton pattern**

As mentioned previously, the printer class uses the singleton pattern so it can be accessed from anywhere in the code without the need to create multiple instances. Several classes in my implementation need to print something.

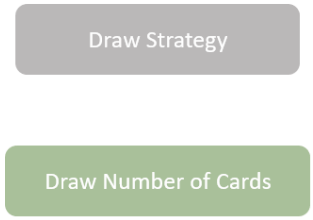
Built-in features

Below we can find all the built-in features in the engine that the developer can choose from.

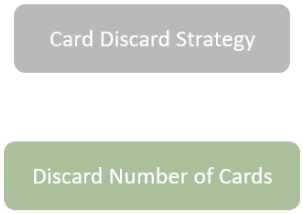
- The cards behaviors the developer can choose from:

- The way the cards can be played over each other:

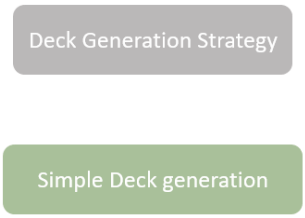
- The way the cards are drawn from the deck, it takes a parameter for the number of cards:



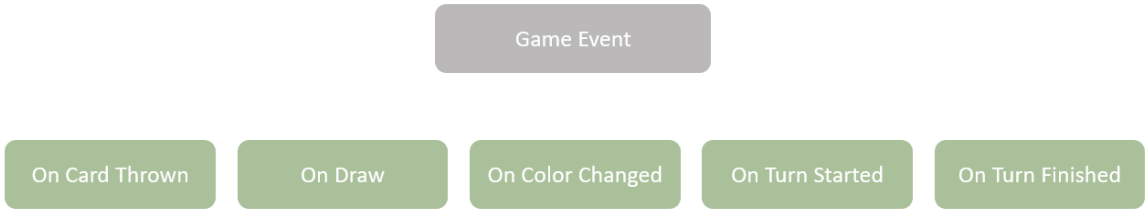
- The way the cards are discarded from player's hand, it takes a parameter for the number of cards:



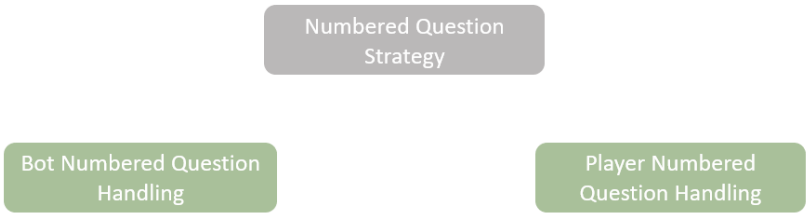
- The way the deck is generated, what cards it contains, how I's initially shuffled.



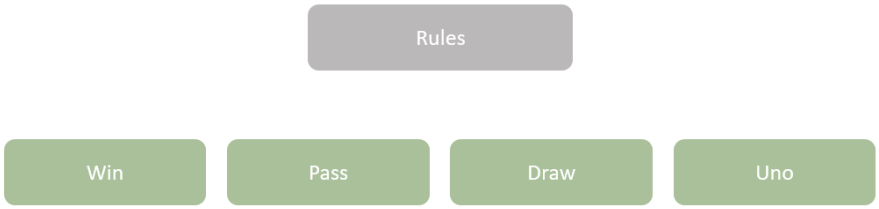
- The important events in the game the developer can add listeners to:



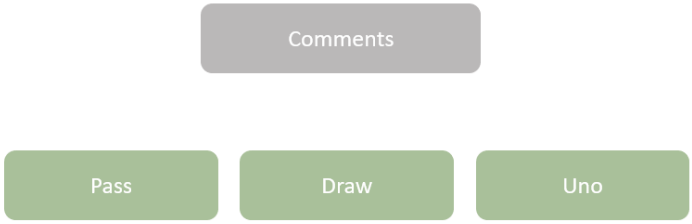
- The different ways to answer a multiple-choice question:



- The rules the developer can choose from:



- The comments the developer can choose from:



Future work

In this section, I will talk about modifications I would have made if I had more time. Please check the list below:

- The factory pattern might be utilized to create different cards.
- Try to find another way to implement playing the player's turn, since it might get complicated if we add a lot of rules and comments. This might be done by utilizing the decorator pattern or finding a whole new way.
- The player class may have a superclass called Person that contains the name of the players and some general fields. This might be necessary so in the future if there is a person who is only watching the game and not playing, his instance would be instantiated from another type than Player, but it will still extend Person.
- Implementing the score for each card. I made it extendible so the developer can create the score calculating mechanism. But it is not already fully implemented for him.
- The visitor pattern might be used to calculate the score of each player.
- Give the differences between the real player and the bot a second thought.