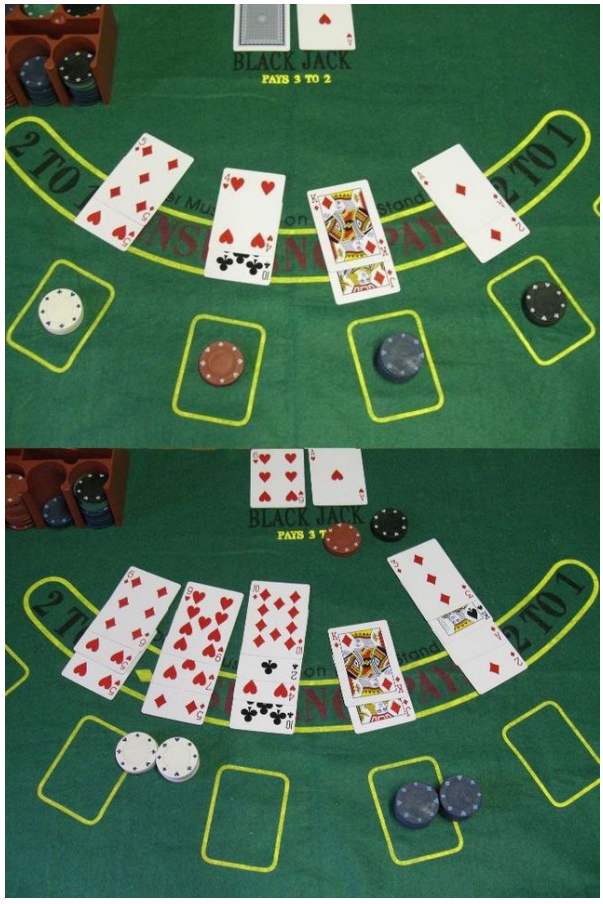
객체지향 프로그래밍

사례 학습 - 블랙잭 카드놀이 소프트웨어 설계 및 구현

소프트웨어 창작에 장인이 되는 건 소설가, 화가, 건축설계사, 교량설계사의 그것과 다를 바 없다. 끊임없는 생각과 습작을 통한 훈련에 시간과 노력을 투자해야 한다. 소설, 그림, 건물, 교량, 무엇이든 바로 제작에 돌입하진 않는다. 제작에 앞서 설계를 먼저 한다. 건물의 경우 설계도인 청사진을 먼저 완성한다. 건물 시공자는 완성된 설계도에 따라 건물을 짓는다. 설계도 없는 건물을 상상해본 적 있는가? 소프트웨어도 마찬가지이다. 코딩을 하기 전에 설계부터 완성해야 한다. 설계가 소프트웨어의 품질을 좌우한다. 그런데 소프트웨어 설계의 장인이 되기 위해서는 다른 분야와 마찬가지로 고도의 훈련이 필요하다.

건축에도 아키텍처가 있고 컴퓨터 하드웨어도 아키텍처가 있듯이, 소프트웨어도 아키텍처가 있다. 코딩을 하기 전에 설계도인 소프트웨어 아키텍처를 먼저 결정해야 한다. 이 장에서는 객체지향 프로그래밍 방식으로 소프트웨어를 만들 때 많이 사용하는 MVC 아키텍처 기반으로 프로그램을 설계하고 작성하는 방법을 블랙잭 카드게임 프로그램의 사례를 통하여 간단히 소개한다.

1. 블랙잭 카드놀이



블랙잭Blackjack은 카지노에서 가장 인기있는 카드놀이 중 하나이다. 21을 넘지 않는 한도 내에서 딜러와 겨루어 점수가 높으면 이기는 놀이다. 컴퓨터가 딜러 역할을 하고 참가자 1인과 표준 입출력창에서 블랙잭 게임을 할 수 있는 프로그램을 작성해보자.

블랙잭 카드놀이 규칙

- 딜러가 카드 두 장을 손님과 자신에게 각각 한장씩 교대로 나누어 준다.
- 딜러의 첫째 카드는 보여주지 않는다.
- 카드의 합이 딜러보다 먼저 21이 되거나, 딜러보다 21에 가깝게 되면 이기고, 카드를 더 받았는데 21을 초과bust하면 진다.
- 먼저 받은 카드 두 장의 합이 21에 못 미치면 원하는 만큼 21이 넘지 않는 한도내에서 한 장씩 더 요청할 수 있다.
- 딜러는 카드의 합이 16 이하이면 무조건 한 장을 더 받아야 하고, 17 이상이면 더 이상 받을 수 없다.
- 딜러의 카드와 합이 같으면 비긴다.

- A(에이스) 카드는 1 또는 11로 취급할 수 있고, 10, J, Q, K는 모두 10으로 계산한다.
- 처음 받은 카드 두 장이 A와 10, J, Q, K 중의 하나로 합이 21이 되면 블랙잭blackjack으로 무조건 이긴다.

요구사항

- 프로그램이 시작하면 `Welcome to SMaSH Casino!` 라는 메시지를 프린트 한다.
- 카드는 1벌(52장)을 잘 섞어서 사용한다. 다 쓰면 1벌 전체를 새로 만들어 섞어서 사용한다.
- 카드는 처음 2장씩 나누어 주는데, 손님, 딜러, 손님, 딜러 순으로 나누어주고, 딜러의 카드는 한장만 보여준다.
- 펼쳐진 카드는 출력할 때 `Spade.J` 와 같은 방식으로 프린트한다. 보여주지 않는 카드는 `xxxxx.xx`로 프린트 한다.
- 처음 2장씩 나누어 준 후, 다음과 같이 출력 창에 프린트해야 한다. 딜러의 이름은 `Dealer`이고, 손님의 이름이 `Pooh`이면,

```
Dealer : Heart.7 XXX
Pooh : Spade.A Diamond.8
```

- 손님은 점수가 21 미만인 경우 카드를 추가로 요청할 수 있다. 딜러는 실행창에 다음과 같이 추가 카드를 원하는지 여부를 물어봐야 하며 손님은 영문의 `o`(예) 또는 `x`(아니오)로 의사를 표시한다. 대문자인 `O`와 `X`도 소문자와 같이 처리할 수 있어야 한다.

```
Hit? (o/x)
```

- 추가로 카드를 받으면 다음과 같이 받은 카드를 모두 프린트 해줘야 한다.

```
Pooh : Spade.A Diamond.8 Heart.2
```

- 딜러는 카드의 합이 16 이하이면 카드를 1장 무조건 받아야 하며, 16을 넘으면 더 이상 받을 수 없다.
- A(에이스)는 1 또는 11 중 하나를 유리한 쪽으로 선택할 수 있어야 한다.
- 손님이 이기면 `Pooh wins.`를 프린트하고 다음 라운드로 넘어간다. 그런데 블랙잭으로 이기면 `Blackjack! Pooh wins.`를 프린트하고, 딜러의 버스트로 이기면 `Dealer busts!`를 프린트한다.
- 손님이 지면 `Pooh loses.`를 프린트하는데, 손님의 버스트로 지면 다음과 같이 프린트 한다.

```
Pooh busts!
```

- 비기면 `We draw.`를 프린트한다.
- 손님이 블랙잭으로 이긴 경우를 제외하고 라운드의 종료와 함께 딜러의 카드를 모두 보여준다.
- 점수 칩의 개수를 나타내며 0에서 시작하여 매 라운드마다 손님이 이기면 1 증가하고, 지면 1 감소한다. 블랙잭으로 이기면 2 증가한다. 딜러는 블랙잭으로 이겨도 보너스가 없다. 점수는 매 라운드마다 다음과 같이 표시한다.

```
Pooh has 6 Chips.
```

- 매 라운드 마다 계속할지 물어봐야 하며 손님은 `o`(예) 또는 `x`(아니오)로 의사를 표시한다.

```
Play more, Pooh? (o/x)
```

- 매 라운드 사이의 구분을 위해서 새 게임은 `== new round ==` 로 시작을 표시한다.
- 게임을 마치면 다음과 같은 메시지를 프린트한다.

```
Bye, Pooh!
```

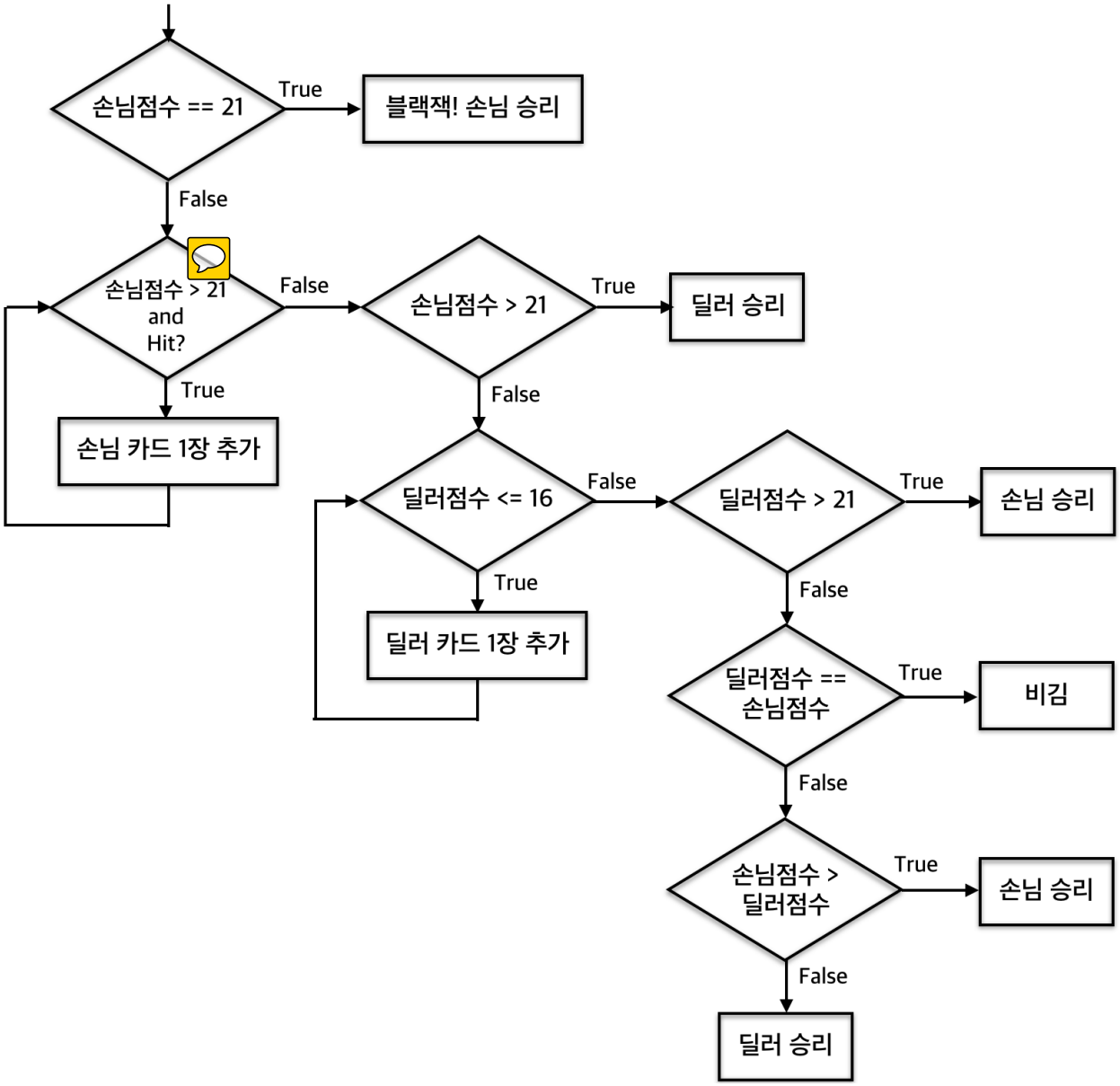
실행사례

```
$ python3 blackjack.py
Welcome to SMaSH Casino!
Enter your name : Pooh
== new game ==
Smavi : Spade.7 XXX
Pooh : Heart.4 Heart.2
Pooh: Hit?(o/x) o
Pooh : Heart.4 Heart.2 Clover.5
Pooh: Hit?(o/x) o
Pooh : Heart.4 Heart.2 Clover.5 Diamond.10
Pooh wins.
Pooh has 1 chips.
Smavi : Spade.7 Clover.9 Spade.A
Play more, Pooh? (o/x) o
== new game ==
Smavi : Diamond.9 XXX
Pooh : Spade.4 Spade.5
Pooh: Hit?(o/x) o
Pooh : Spade.4 Spade.5 Spade.J
Pooh: Hit?(o/x) x
Smavi busts.
Pooh has 2 chips.
Smavi : Diamond.9 Heart.6 Diamond.K
Play more, Pooh? (o/x) o
== new game ==
Smavi : Heart.5 XXX
Pooh : Clover.K Diamond.J
Pooh: Hit?(o/x) x
Smavi busts.
Pooh has 3 chips.
Smavi : Heart.5 Diamond.8 Heart.3 Heart.Q
Play more, Pooh? (o/x) o
== new game ==
Smavi : Clover.6 XXX
Pooh : Diamond.Q Spade.10
Pooh: Hit?(o/x) x
Pooh wins.
Pooh has 4 chips.
Smavi : Clover.6 Heart.9 Clover.3
Play more, Pooh? (o/x) o
== new game ==
Smavi : Clover.Q XXX
Pooh : Clover.2 Clover.J
Pooh: Hit?(o/x) o
Pooh : Clover.2 Clover.J Diamond.6
Pooh: Hit?(o/x) x
Smavi busts.
Pooh has 5 chips.
Smavi : Clover.Q Diamond.3 Diamond.2
Spade.K
Play more, Pooh? (o/x) o
== new game ==
Smavi : Heart.J XXX
Pooh : Heart.K Spade.8
Pooh: Hit?(o/x) x
Pooh loses.
Pooh has 4 chips.
Smavi : Heart.J Clover.10
Play more, Pooh? (o/x) o
== new game ==
Smavi : Diamond.A XXX
Pooh : Heart.A Clover.7
Pooh: Hit?(o/x) o
Pooh : Heart.A Clover.7 Spade.2
Pooh: Hit?(o/x) x
Pooh wins.
Pooh has 5 chips.
Smavi : Diamond.A Diamond.7
Play more, Pooh? (o/x) o
== new game ==
Smavi : Clover.4 XXX
Pooh : Clover.A Spade.9
Pooh: Hit?(o/x) x
Pooh wins.
Pooh has 6 chips.
Smavi : Clover.4 Spade.3 Diamond.5 Spade.
6
Play more, Pooh? (o/x) o
== new game ==
Smavi : Heart.10 XXX
Pooh : Diamond.4 Clover.8
Pooh: Hit?(o/x) o
Pooh : Diamond.4 Clover.8 Heart.8
Pooh: Hit?(o/x) x
We draw.
Smavi : Heart.10 Spade.Q
Play more, Pooh? (o/x) o
== new game ==
Smavi : Diamond.6 XXX
Pooh : Heart.7 Diamond.2
Pooh: Hit?(o/x) o
Pooh : Heart.7 Diamond.2 Diamond.J
Pooh: Hit?(o/x) x
Smavi busts.
Pooh has 7 chips.
Smavi : Diamond.6 Spade.10 Clover.7
Play more, Pooh? (o/x) o
== new game ==
Smavi : Spade.5 XXX
Pooh : Clover.5 Clover.4
Pooh: Hit?(o/x) o
Pooh : Clover.5 Clover.4 Heart.10
Pooh: Hit?(o/x) x
We draw.
Smavi : Spade.5 Heart.3 Heart.A
Play more, Pooh? (o/x) o
== new game ==
Smavi : Diamond.10 XXX
Pooh : Heart.8 Heart.9
Pooh: Hit?(o/x) o
Pooh : Heart.8 Heart.9 Spade.8
Pooh busts.
Pooh loses.
Pooh has 6 chips.
Smavi : Diamond.10 Clover.Q
Play more, Pooh? (o/x) x
Bye, Pooh!
```

블랙잭 알고리즘

손님이 원하는 한, 다음 라운드를 반복한다.

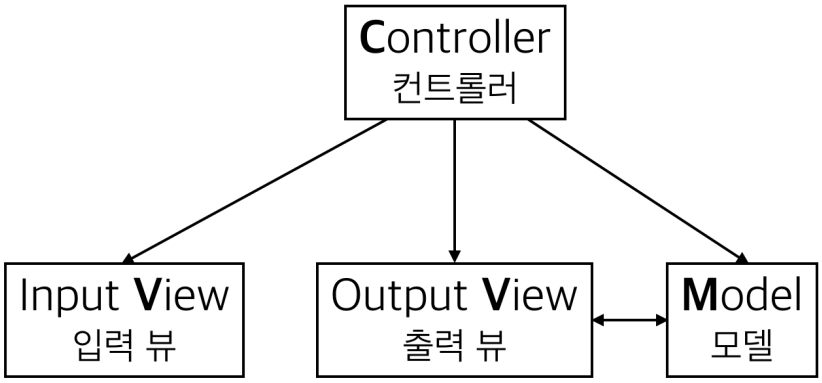
1. 손님, 딜러, 손님, 딜러 순으로 카드를 총 2장씩 배분한다.
2. 손님의 카드는 모두 보여주고, 딜러의 카드는 한장만 보여준다.
3. 손님과 딜러의 카드 두 장의 정수를 각각 계산한다.
4. 다음 흐름차트에 따라서 게임을 진행한다.



5. 더 할지 손님에게 물어봐서 그만 하길 원하면 끝내고, 더 하길 원하면 1번으로 돌아가서 계속한다.

2. MVC 아키텍처 기반 소프트웨어 설계 및 구현

지금까지 공부한 프로그램을 보면 명령창에서 키보드 입력을 받아서 계산을 한 뒤 결과값을 명령창에 보여주는 아주 간단한 형태로 작동이 된다. 이 절에서는 소프트웨어를 역할 별로 부품으로 나누어 서로 소통하면서 임무를 수행하도록 설계하는 MVC(Model-View-Controller) 아키텍처를 공부해보자. 소프트웨어의 MVC 아키텍처를 간단하게 그림으로 표현하면 다음 그림과 같다.



MVC 아키텍처의 각 부품은 다음과 같은 역할을 맡는다.

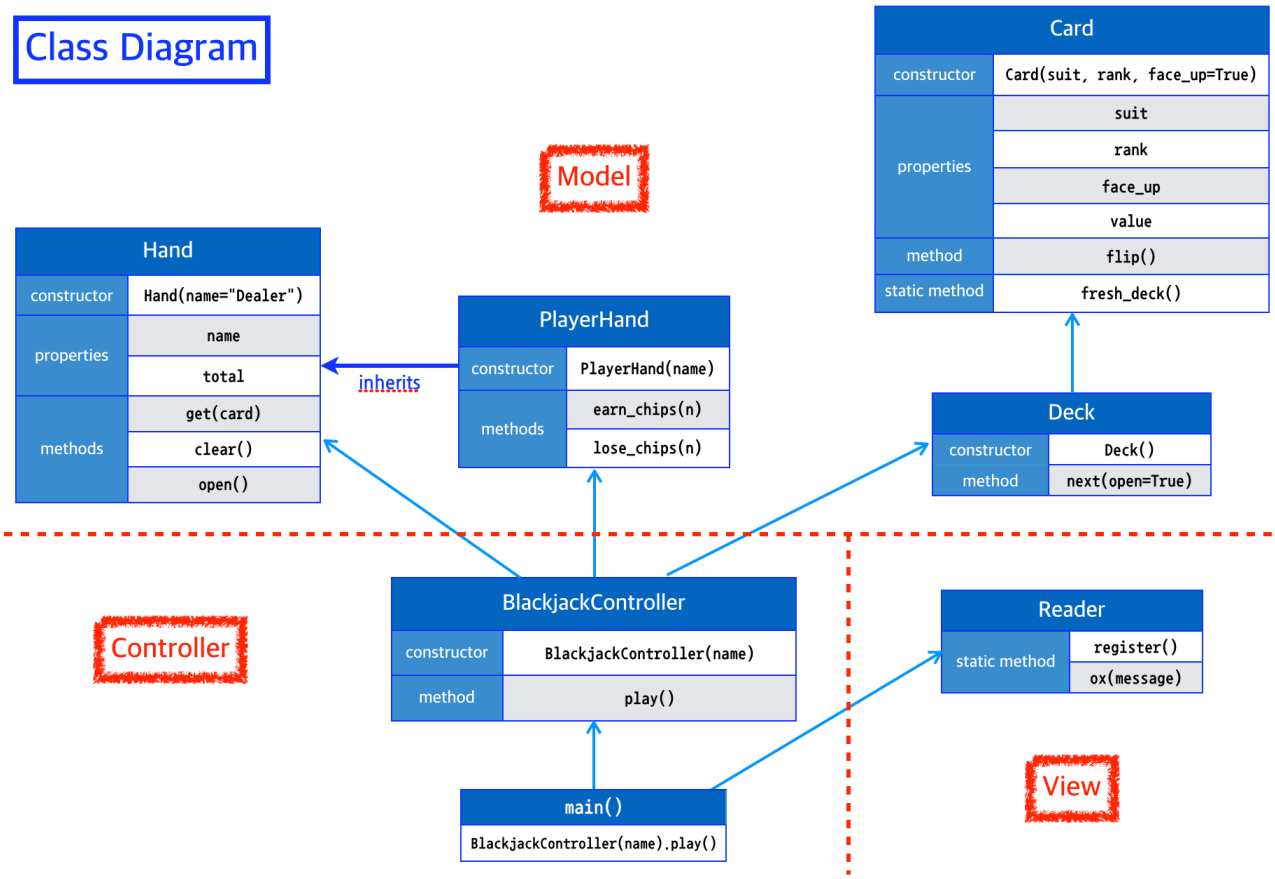
- 컨트롤러Controller는 프로그램의 흐름(실행순서)을 제어한다.
- 입/출력 뷰View은 외부와의 의사소통을 담당한다.
- 모델Model은 문제를 모델링하고 해답을 계산하는 역할을 한다.

위 그림에서 화살표의 의미를 하나씩 차례로 새겨보자.

- 컨트롤러는 입력 뷰에 메시지를 보내 데이터를 요청한다.
- 컨트롤러는 그 데이터를 가지고 모델에 계산을 요청한다.
- 컨트롤러는 출력 뷰에게 계산결과를 보여주라고 요청한다.
- 경우에 따라서 모델이 직접 출력 뷰에 계산결과를 보여주라고 요청할 수도 있고, 출력 뷰가 능동적으로 모델에서 계산 결과를 가져다 보여줄 수도 있다.

클래스 다이어그램

블랙잭 카드놀이의 완성된 설계도인 클래스 다이어그램을 미리 구경해보자.



클래스 다이어그램은 소프트웨어의 설계도라고 할 수 있다. 점선으로 세 구역으로 나뉘는데, 위 쪽에 모델, 아래 왼쪽에 컨트롤러, 아래 오른쪽에 뷰에 해당되는 클래스들이 모여 있다. 클래스 다이어그램에서 각 테이블은 클래스를 나타내는데, 클래스의 이름과 함께, 겉으로 드러나는 속성변수와 메소드 정보를 기술해 놓았다. 이 클래스 다이어그램을 참고하면 각 클래스를 독립적으로 설계하고 구현할 수 있다. 이제 이 클래스 다이어그램을 구성하고 있는 각 클래스들이 구현 전략을 어떻게 세우고, 이를 바탕으로 어떻게 구현하는지 하나씩 모델, 뷰, 컨트롤러 순으로 자세히 살펴보자.

Card

앞 장에서 공부한 Card 클래스를 그대로 사용하되, 카드의 점수를 기록해두는 비공개 속성을 추가한다[1].

```
def __init__(self, suit, rank, face_up=True):
    if suit in Card.__suits and rank in Card.__ranks:
        self.__suit = suit
        self.__rank = rank
        self.__face_up = face_up
    else:
        print("Error: Not a valid card")
    self.__value = Card.__ranks.index(self.__rank) + 1
    if self.__value > 10:
        self.__value = 10
```

"J", "Q", "K"는 모두 10점으로 매긴다. "A"는 경우에 따라서 1점 또는 11점 선택하여 적용할 수 있으나 일단 1점 그대로 둔다.

Card			
class	attributes	self.__suits	("Diamond", "Heart", "Spade", "Clover")
		self.__ranks	("A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K")
	static method	fresh_deck()	returns a brand-new deck of shuffled cards with all face down
object	properties	suit	its suit value in Card.__suits
		rank	its rank value in Card.__ranks
		face_up	its face_up value : True or False
		value	its face value (according to blackjack rule)
	constructor	Card(suit, rank, face_up=True)	creates a playing card object arguments: suit -- must be in Card.__suits rank -- must be in Card.__ranks face_up -- True or False (default True)
	method	flip()	flips itself

[1] 사실 Card 클래스를 그대로 두고 이를 상속받고 self.__value 속성을 추가하하는 Blackjack 클래스를 따로 만드는 것이 객체지향 설계 방식을 잘 따르는 것이지만 이는 뒤에서 따로 고민하기로 하고 지금은 그냥 넘어가자.

Deck

카드 한 벌deck을 형상화하여 관리하는 객체를 만드는 클래스는 Deck 이다.

Deck			
object	atrribute	self.__deck	Deck object
	constructor	Deck()	creates a deck object consisting of 52 cards shuffled
	method	next(open=True)	removes a card from deck and returns the card with its face up if open == True, or with its face down if open == False

Card.fresh_deck() 메소드를 호출하면 잘 섞인 카드 한 벌을 언제나 만들 수 있다. self.__deck은 비공개 속성변수로 게임에 사용할 섞인 카드들을 리스트로 갖고 있어야 하며, Deck 객체를 처음 만들 때 다음과 같이 Card.fresh_deck() 메소드를 호출하여 받은 Card 객체 리스트로 초기화한다.

```
def __init__(self):
    self.__deck = Card.fresh_deck()
    print("<< A brand-new deck of card! >>")
```

Deck 객체 메소드는 next() 이다. 이 메소드를 호출하면 self.__deck에서 카드 객체를 하나 뽑아서 내준다.

```
def next(self, open=True):
    if self.__deck == []:
        self.__deck = Card.fresh_deck()
        print("<< A brand-new deck of card! >>")
    card = self.__deck.pop()
    if open :
        card.flip()
    return card
```

그런데 카드를 다 쓰고 없으면, Card.fresh_deck() 메소드를 호출하여 카드 한 벌을 새로 받아온다. 카드를 줄 때 앞면으로 주고 싶을 경우도 있고 뒷면으로 주고 싶을 경우도 있다. 이 정보는 next() 메소드의 인수(open)로 전달을 하는데, 이 인수의 기본 값은 True로 정하였으므로 next()와 같이 언급이 없으면 True로 취급하여 카드를 뒤집어 앞면으로 주고, next(False)와 같이 명시적으로 요청을 하면 그대로 뒷면으로 내준다.

위의 클래스 다이어그램을 보면, Deck 클래스에서 Card 클래스로 화살표가 그어져 있다. Deck() 객체는 Card.fresh_deck() 을 호출하여 Card 클래스를 사용하니, 그 사용 관계를 표시한 화살표이다.

Hand

Hand 클래스는 각 게임 참여자가 받아서 갖고 있는 카드를 형상화한다.

Hand			
object	constructor	Hand(name="Dealer")	creates player/dealer's empty hand argument: name -- player's name in string (default:
	attributes	self.__name	its name : either player's name or 'Dealer'
		self.__hand	list of Card objects in its hand
	properties	name	its name : either player's name or 'Dealer'
		total	the total value of its hand
	methods	get(card)	gets a card from deck and puts the card into its hand
		clear()	empties its hand
		open()	turns all of its hand's cards' faces up

Hand 객체가 가지고 있어야 하는 속성은 자신의 이름 self.__name과 소지 카드 목록 self.__hand인데 모두 비공개 속성으로 다음과 같이 생성메소드에서 초기 값을 설정한다. 전자는 카드 소지자의 이름으로 인수로 받은 문자열로 설정하고, 후자는 빈 리스트로 설정한다. name 인수를 생략하면 자동으로 'Dealer'로 설정한다.

```
def __init__(self, name="Dealer"):
    self.__name = name
    self.__hand = []
```

Hand 객체의 외부에 공개할 정보인 이름 name과 소지 카드의 액면점수 total은 프로퍼티로 다음과 같이 정의한다. 프로퍼티 name은 self.__name 문자열을 그대로 내주게 하고, 프로퍼티 total은 갖고 있는 카드의 액면 점수를 계산해준다. 'A' 카드의 경우 1 또는 11을 선택적으로 부여할 수 있는데, 일단 11로 계산한 다음 21이 넘는 경우 10을 빼서 1로 계산한다. 'A' 카드를 두 장 이상 갖고 있는 경우도 고려하여 코드를 작성했음을 주의 깊게 보자.

```
@property
def name(self):
    return self.__name

@property
def total(self):
    point = 0
    number_of_ace = 0
    for card in self.__hand:
        if card.rank == 'A':
            point += 11
            number_of_ace += 1
        else:
            point += card.value
    while point > 21 and number_of_ace > 0:
        point -= 10
        number_of_ace -= 1
    return point
```

Hand 객체의 메소드는 다음과 같이 정의한다. get(card)를 호출하면 건네받은 card 객체를 자신의 카드 리스트에 추가한다. clear()는 카드 리스트를 비우는 메소드이고, open() 메소드는 자신 갖고 있는 카드를 모두 압면으로 뒤집는 메소드이다.

```
def get(self, card):
   self.__hand.append(card)

def clear(self):
   self.__hand = []

def open(self):
   for card in self.__hand:
        if not card.face_up:
            card.flip()
```

PlayerHand

Hand 클래스는 손님과 딜러가 공통으로 사용할 수 있지만, 손님의 경우 자신이 따거나 잃은 칩의 개수를 기록해두고 알려주는 기능이 추가로 있어야 한다. Hand 클래스가 갖고 있는 속성과 메소드는 그대로 유지한 채, 새 속성과 메소드를 추가하고 싶으면 **상속**inheritance 기능을 활용하면 된다. 추가해야 할 속성과 메소드가 다음과 같으므로, 손님 전용 클래스 PlayerHand 클래스는 다음과 같이 설계하고,

PlayerHand			
object	constructor	PlayerHand(name)	creates player's empty hand with the capability of counting chips it owns argument: name -- player' name in string
	attribute	self.__chips	the number of chips it has (initial value = 0)
	method	earn_chips(n)	increases the number of chips by n
		lose_chips(n)	decreases the number of chips by n

PlayerHand 클래스를 Hand 클래스를 상속받고 다음과 같이 작성한다.

```
class PlayerHand(Hand):
   def __init__(self, name):
       super().__init__(name)
        self.__chips = 0

    def earn_chips(self, n):
       self.__chips += n
        print("Your have", self.__chips, "chips.")

    def lose_chips(self, n):
       self.__chips -= n
        print("Your have", self.__chips, "chips.")
```

상속받을 클래스는 클래스 선언 부분에서 파라미터로 상속받는 클래스 이름을 명시한다. 이 경우 새로 선언하는 PlayerHand 클래스가 Hand 클래스를 상속받으므로 class PlayerHand(Hand) 라고 기술한 것이다. 이 때, PlayerHand 클래스를 **하위클래스**subclass**라** 하고, Hand 클래스를 **상위클래스**superclass라고 한다. 하위클래스에서 상위클래스를 지칭하려면, 그냥 간단히 super()라고 하면 된다. 생성메소드에서 상위클래스의 생성메소드를 호출하여 실행하고, 자신의 속성변수인 self.__chips의 초기값을 0으로 설정한다. 남은 두 메소드는 각각 self.__chips 속성변수의 값을 인수만큼 줄이거나 늘리는 역할을 한다. 이와 같이 상속을 이용하면 이미 완성되어 있는 클래스를 재사용하고 추가할 속성이나 메소드만 추가하면 되므로 편리하다. 객체지향 프로그래밍에서 가장 강력한 기능 중의 하나이므로 효과적으로 사용하면 좋다.

위의 클래스 다이어그램을 보면 PlayerHand에서 Hand로 다른 화살표와는 모양이 다른 화살표가 있다. 이 화살표는 상속관계를 나타낸다.

Reader

Hand 클래스는 MVC 아키텍처에서 뷰 역할을 하는 클래스이다. 사용자의 이름을 표준입력창에서 받아서 내주는 `register()` 함수와 사용자의 의사를 대소문자 구분없이 o 또는 x로 표준입력창에서 받아서 소문자로 일관되게 내주는 `ox(message)` 함수를 Reader 클래스 소속의 정적 메소드로 선언하였다.

Reader		
static method	`register()`	gets player's name and returns it (string)
	`ox(message)`	returns True if player inputs 'o' or 'O', False if player inputs 'x' or 'X'

정적메소드임을 밝히는 `@staticmethod` 인식표만 앞에 다음과 같이 붙여주면 된다.

```
class Reader:
    @staticmethod
    def register():
        return input("Enter your name : ")

    @staticmethod
    def ox(message):
        response = input(message).lower()
        while not (response == 'o' or response == 'x'):
            response = input(message).lower()
        return response == 'o'
```

BlackjackController

마지막으로 BlackjackController 클래스는 컨트롤러 역할을 하는 클래스로 한 라운드의 게임 진행을 진행하는 역할을 하는 play() 메소드를 장착하고 있다.

BlackjackController			
object	constructor	BlackjackController(name)	creates player/dealer's empty hand and a deck of cards argument: name -- player' name in string (default: 'Dealer')
	attributes	self.__player	PlayerHand object
		self.__dealer	Dealer object
		self.__deck	Deck object
	method	play()	plays a round of blackjack game

게임의 등장인물은 손님인 PlayerHand 객체, 딜러인 Hand 객체, 그리고 Deck 객체가 있다. 클래스 다이어그램을 보면 BlackjackController 클래스에서 이 세 클래스로 나가는 화살표가 있는데 이 화살표도 사용관계를 나타내고 있다. BlackjackController 객체를 생성하는 메소드는 다음과 같이 작성한다.

```
def __init__(self, name):
    self.__player = PlayerHand(name)
    self.__dealer = Hand()
    self.__dealer = Deck()
```

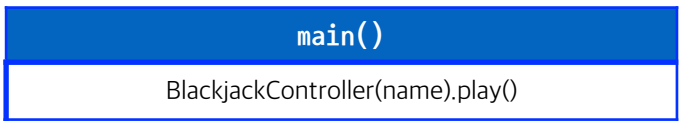
등장인물이 되는 세 객체를 생성한 다음, 각각을 비공개 속성변수로 각각 지정한다.

play() 메소드는 아래와 같이 정의한다. 비공개 속성변수인 self.__player, self.__dealer, self.__dealer 는 메소드 내에서 자주 사용하므로 짧은 이름으로 재지정하였다. 알고리즘은 이전 함수와 똑 같으니 세 등장인물이 각자의 메소드를 호출하면서 진행하는 과정을 잘 관찰하면서 객체지향 프로그램의 작동 원리를 이해해보기 바란다.

```
def play(self):
    print("== new game ==")
    player = self.__player
    dealer = self.__dealer
    deck = self.__deck
    player.get(deck.next())
    dealer.get(deck.next())
    player.get(deck.next())
    dealer.get(deck.next(open=False))
    print("Dealer :", dealer)
    print(player.name, ":", player)
    if player.total == 21:
        print("Blackjack!", player.name, "wins.")
        player.earn_chips(2)
    else:
        while player.total < 21 and \
              Reader.ox(player.name + ": Hit?(o/x) "):
            player.get(deck.next())
            print(player.name, ":", player)
        if player.total > 21:
            print(player.name, "busts!")
            player.lose_chips(1)
        else:
            while dealer.total <= 16:
                dealer.get(deck.next())
            if dealer.total > 21:
                print("Dealer busts!")
                player.earn_chips(1)
            elif dealer.total == player.total:
                print("We draw.")
            elif dealer.total > player.total:
                print(player.name, "loses.")
                player.lose_chips(1)
            else:
                print(player.name, "wins.")
                player.earn_chips(1)
        dealer.open()
        print("Dealer :", dealer)
    player.clear()
    dealer.clear()
```

메인 함수

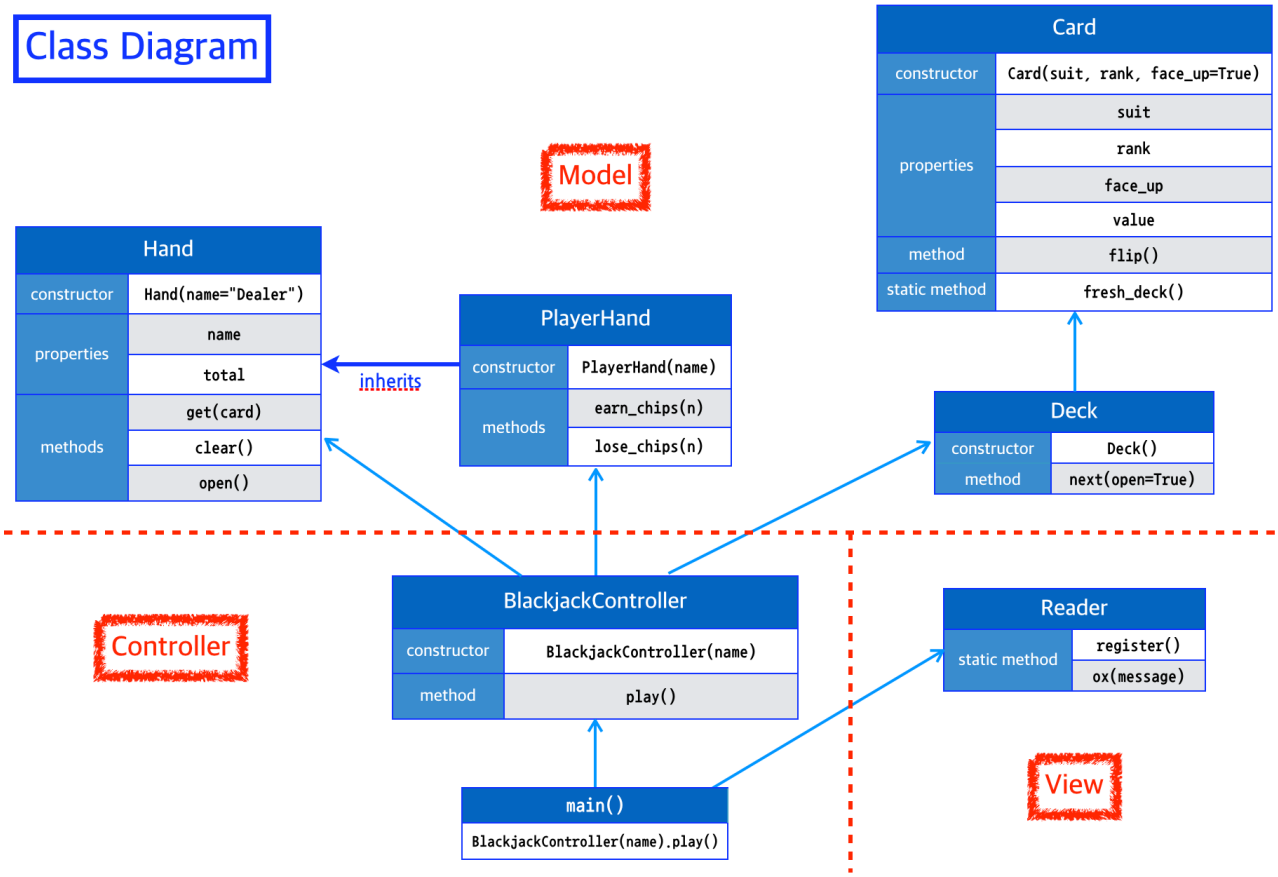
메인 함수의 차이점은 BlackjackController 객체를 생성한 다음, play() 메소드를 호출한다는 점만 다르다.



아래 코드를 잘 관찰하여 차이점을 이해하기 바란다.

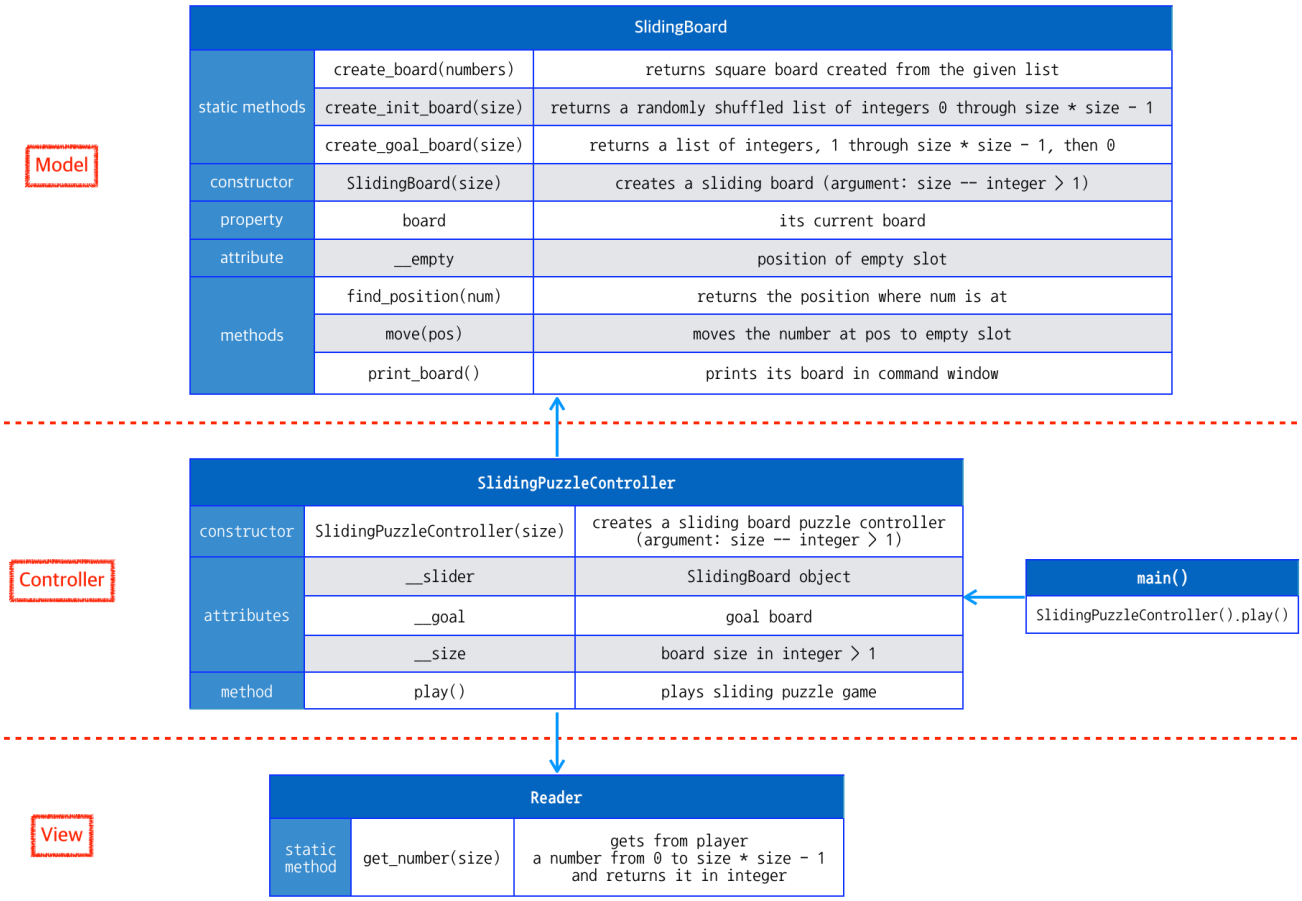
```
def main():
   print("Welcome to SMaSH Casino!")
    name = Reader.register()
    game = BlackjackController(name)
    while True:
        game.play()
        if not Reader.ox("Play more, " + name + "? (o/x) "):
            break
    print("Bye, " + name + "!")
```

클래스 다이어그램을 보면서 코드리뷰를 해보았다. 다시 한번 클래스 다이어그램을 읽고 전체 프로그램의 구조를 음미해보자.



실습

숙제 #3으로 구현한 슬라이딩 퍼즐게임을 다음 클래스 다이어그램에서 설계한 MVC 구조에 맞추어 객체지향으로 재구현하자.



1단계

뷰에 해당하는 Reader 클래스를 먼저 만들어보자. 클래스 소유의 정적메소드로 정의할 것이므로 기존의 함수를 그대로 가져다 @staticmethod 만 다음과 같이 붙여주면 된다.

```
class Reader:
    @staticmethod
    def get_number(size):
        num = input("Type the number you want to move (Type 0 to quit): ")
        while not (num.isdigit() and 0 <= int(num) <= size * size - 1):
            num = input("Type the number you want to move (Type 0 to quit): ")
        return int(num)
```

이제 이 메소드는 어디에서든지 Reader.get_number(4)와 같은 형식으로 호출할 수 있다.

이제 모델에 해당하는 SlidingBoard 클래스를 만들 차례이다.

2단계

이 클래스 내부에 들어갈 메소드를 유형별로 하나씩 작성하자. 게임보드, 해답보드를 중첩리스트로 만들어 내주는 함수들은 모두 클래스 소속의 정적메소드로 정의한다. 따라서 바로 위의 get_number 메소드

와 같은 요령으로 세 함수를 그대로 넣으면 된다. 단, 이제 이 메소드를 호출하려면 어디서든지 반드시 클래스의 이름인 SlidingBoard.을 반드시 앞에 붙여야 한다.

3단계

다음은 생성메소드constructor인 __init__를 만들어보자. 비공개 속성변수로 self.__board와 self.__empty를 여기서 지정해주어야 함을 명심하자. self.__board는 인수로 받은 size 크기로 초기 게임보드를 만들어 설정하고, self.__empty는 게임보드에서 0의 위치좌표를 찾아 설정해야 한다. 그리고 self.__board는 프로퍼티로 공개하자고 하므로 다음과 같이 추가로 프로퍼티로 정의해야 한다.

```
    def __init__(self, size):
        self.__board =
        self.__empty =

    @property
    def board(self):
        return self.__board
```

4단계

남은 메소드 세 개는 모두 객체 소속 메소드이다. 기존 함수를 그대로 쓰되, 객체 소속 메소드이므로 첫 인수로 self를 반드시 추가해주어야 한다. 그리고 board와 empty 변수는 이제 속성변수인 self.__board와 self.__empty로 바꿔주어야 한다. 예를 들어, find_position 함수의 전과 후를 비교해보면 다음과 같다.

함수	def find_position(num, **board**):
	  for i in range(len(**board**)):
	  for j in range(len(**board**)):
	  if num == **board**[i][j]:
	  return (i, j)

메소드	def find_position(**self**, num):
	  for i in range(len(**self.__board**)):
	  for j in range(len(**self.__board**)):
	  if num == **self.__board**[i][j]:
	  return (i, j)

move 메소드를 작성할 때는 조심해야 한다. 함수 버전은 빈칸의 위치좌표와 보드 전체를 함수의 결과로 내주었지만, 메소드의 경우 이 두 값 모두 내줄 필요없이 속성변수를 수정하도록 해야 한다.

5단계

이제 컨트롤러인 SlidingPuzzleController 클래스를 만들어보자. 이 클래스에는 sliding_puzzle 함수가 하던 작업을 도맡아서 하도록 해야한다. 이 게임에 등장하는 게임보드 객체, 게임 종료를 판단하기 위한 해답보드, 보드의 크기 정보를 각각 담고 있을 비공개 속성변수인 self.__slider, self.__goal, self.__size 값은 생성메소드에서 설정한다. 남은 부분은 객체소속 메소드인 play()에 적절히 수정하여 넣으면 된다.

6단계 (마무리)

메인 프로시저는 다음과 같이 작성한다.

```
def main():
    import sys
    size = sys.argv[1]
    if size.isdigit() and int(size) > 1:
        SlidingPuzzleController(int(size)).play()
    else:
        print("Not a proper system argument.")
```

이제 슬라이딩 퍼즐게임을 이전과 똑같이 실행할 수 있는지 실행창에서 확인해보자.

```
$ python3 slidingpuzzleOOP.py 4
```