Lecture 5 - 棋牌类: Black Jack & Chinese Chess

棋牌类解题思路

- 特点 - Clarify
  - 玩家
  - 规则
  - 胜负
  - 积分
- 术语 - Core Object
  - Board
  - Suit
  - Hand
- State - Use Case
  - Initialization (摆盘，洗牌...)
  - Play (下棋，出牌...)
  - Win/Lose check (胜负结算) + Tie (流局)

TicTacToe

```
//Simulator.java
makeMove(1,1)

//TicTacToe.java
public void makeMove(int row, int col) {
  board.makeMove(row, col, currentMove);
  if (board.checkWin()) {
    print (currentMove + " win!");
  } else if (board.isBoardFull()) {
    print("It's a tie");
  }
  changePlayer();
}
```

Chinese Chess

- 特点 - Clarify
  - 玩家
  - 规则
  - 胜负
  - 积分
- 术语 - Core Object
  - Board
  - Suit
  - Hand
- State - Use Case
  - Initialization (摆盘，洗牌...)
  - Play (下棋，出牌...)
  - Win/Lose check (胜负结算) + Tie (流局)

牌类 - Blackjack

- Core Objects - 牌类较为固定的 framework
  - Hand, Player, Dealer, Card, Deck
- Use Case
  - Initialization (摆盘，洗牌...)
    - Join table
    - Place bet
    - Get initial cards
  - Play (下棋，出牌...)
    - Deal
    - Increase bet
    - Stop dealing
  - Win/Lose check (胜负结算) + Tie / Draw (平局)
    - Compare score
    - Take/Lose bets

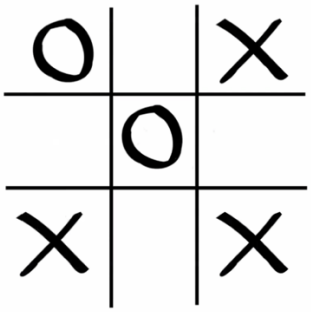
棋牌类解题思路

- 棋牌类的状态：一局棋牌，分为哪些状态（State）？

- Initialization (摆盘，洗牌…)
- Play (下棋，出牌…)
- Win/Lose check (胜负结算) + Tie (流局)

针对棋牌类的状态，来做Use cases

Tic Tac Toe



Clarify

- 玩家： Player之间有什么区别

玩家A：X
玩家B：O

```
currentPlayer = “X”;

changePlayer()
{
    if(currentPlayer.equals(“X”)) currentPlayer = “O”;
    else currentPlayer = “X”;
}
```

Clarify

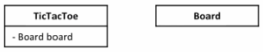
- 什么时候需要Player类？（Player之间还会有什么区别？）

积分

- 规则

对于本题：X always takes the first move
对于本题：3 X 3

Core Object




```
Simulator.java

makeMove(1,1);
```

```
TicTacToe.java

public void makeMove(int row, int col)
{
    board.makeMove(row, col, currentMove);
    if(board.checkWin())
    {
        print(currentMove + " win !");
    }
    else if(board.isBoardFull())
    {
        print("It's a tie");
    }
    changePlayer();
}
```

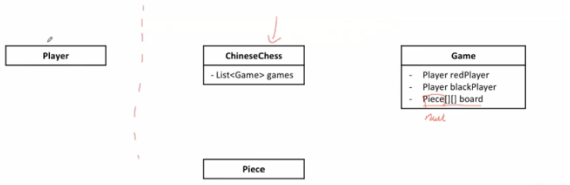
- 胜负

- 如何判定平局？

→ Solution 1: 如果下的步数超过一定数量，判定平局
Solution 2: 电脑判定，如果双方一直在走重复的步子，判定平局
Solution 3: 如果双方选手都要求平局，判断平局

solution1 简单很多，最好用这个

Core Object



Use case

- Initialization (摆盘，洗牌…)

- Join game
- Set up game

Use case

- Play (下棋，出牌…)

- Make move
- Change player

Use case

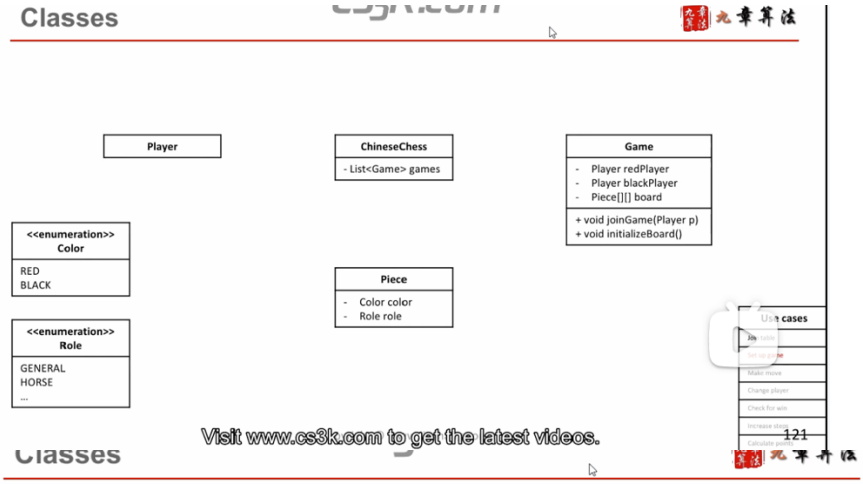
- Win/Lose check (胜负结算) + Tie / Draw (平局)

- Check for win
- Increase steps
- Calculate points

Classes

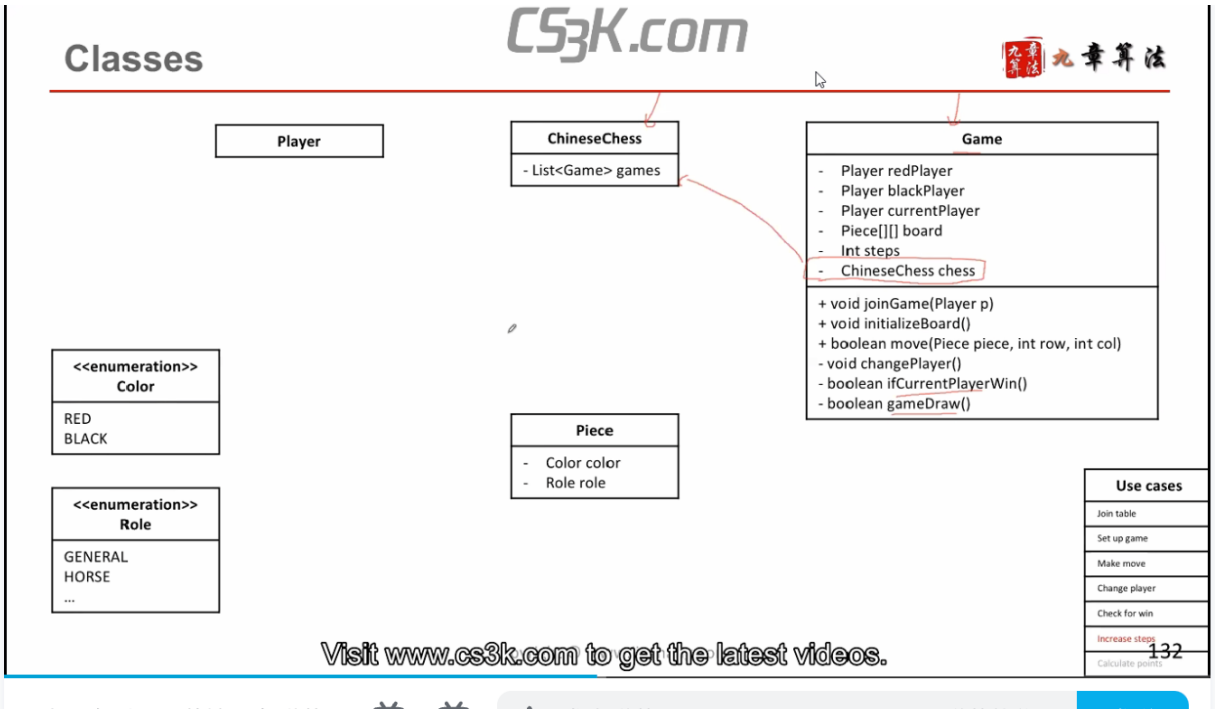
- Set up game

Initialize the board with all pieces placed at the right place.



- Make move

- Determine which player should take the move
- Check if the move if valid, if yes, return true and make the move, if not return false



Classes

- Calculate points

If current player wins, reward current player and take one point off from other one.

blackjack

Clarify

- 对于本题：

- 无人数上限
- 每桌有Fixed dealer
- 牌永远够用
- Dealer的筹码永远够用
- 每个人有同样的初始筹码

Use case

- Initialization (摆盘，洗牌…)

- Join table
- Place bet
- Get initial cards

Use case

- Play (下棋，出牌…)

- Deal
- Increase bet
- Stop dealing

Use case

- Win/Lose check (胜负结算) + Tie / Draw (平局)

- Compare score
- Take/Lose bets

Classes

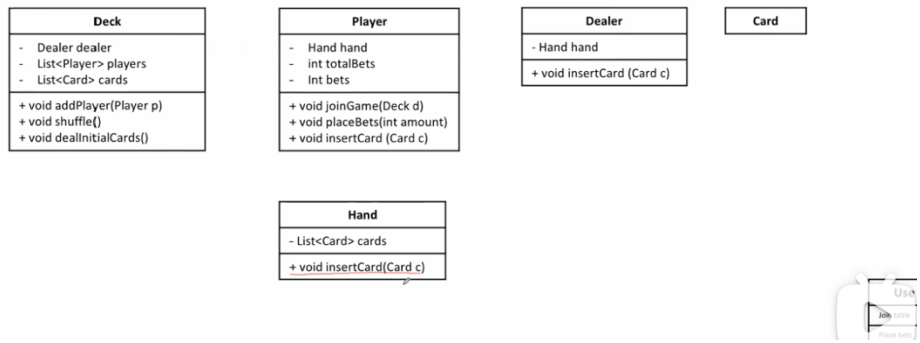
180

只有棋牌类才加 player 这个类

Get initial hands

- Each player and dealer get 2 initial cards

Classes



Deal

- Player decides whether they want to get another card

Classes

Deck
- Dealer dealer - List<Player> players - List<Card> cards
+ void addPlayer(Player p) + void shuffle() + void dealInitialCards() + Card dealNextCard()

Player
- Hand hand - int totalBets - Int bets - Deck d
+ void joinGame(Deck d) + void placeBets(int amount) + void insertCard(Card c) + void dealNextCard()

Dealer
- Hand hand
+ void insertCard (Card c)

Card

Hand
- List<Card> cards
+ void insertCard(Card c)

Use cases
Join table
Place bets
Get initial cards
Deal
Stop dealing
Compare scores
Take/Lose bets

Classes

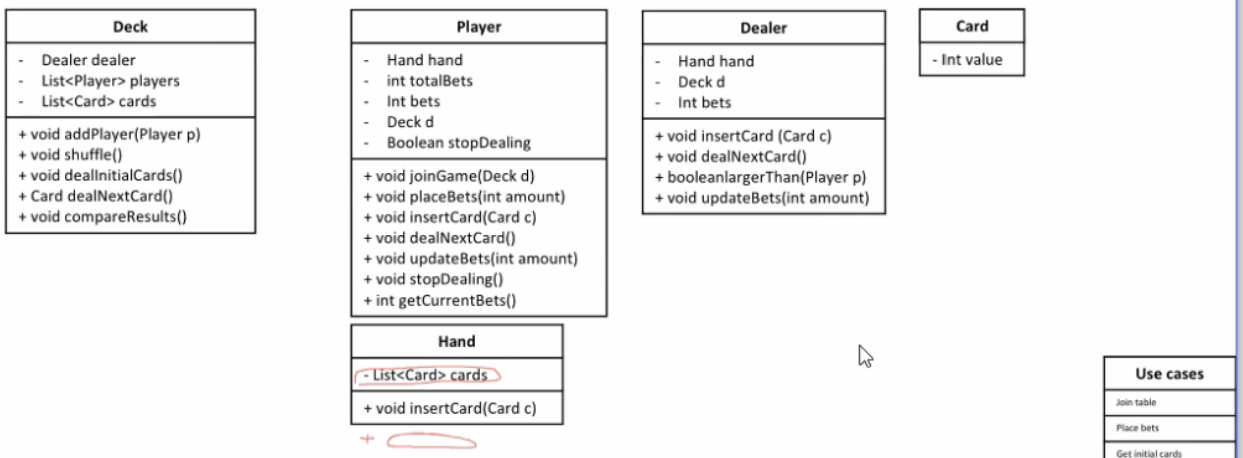
Simulator.java

```
Player player_1 = new Player();

player_1.dealNextCard();


public void dealNextCard()
{
    Card nextCard = deck.dealNextCard();
    insertCard(nextCard);
}
```

Classes



```
Deck.compareResult();

for(Player player : players)
{
    int currentBets = player.getCurrentBets();
    if(dealer.largerThan(player))
    {
        dealer.updateBets(currentBets);
        player.updateBets(-currentBets);
    }
    else{
        dealer.updateBets(-currentBets);
        player.updateBets(currentBets);
    }
}
```

棋牌类总结

- Clarify : 玩家，规则，胜负，积分
- Core object: Hand, Board, Deck/Table, Suit, …
- Use cases: Initialization / Play / Checkout

- 对于牌类，需要从Player的角度出发

Design pattern 总结

- Singleton
- Strategy
- Adapter
- State
- Decorator
- Factory

Singleton

- 用途：

考虑你设计的东西，是否应该只有一个实例

- ElevatorSystem vs. Elevator
- 象棋大厅 vs. 象棋 / Deck / Table
- Kindle 内部的 ReaderFactory

Singleton

九章算法

面试中：

不需要一上来就考虑Singleton.

做完class diagram之后：

- So I was thinking maybe we can apply singleton pattern to this ReaderFactory as well, because…
- Do you think there should be only one instance of the Elevator System?

State

- 出现频率不高
- 特别适合于特殊类型的题目

e.g. Management类型 -> Parking Lot

State: OPEN v.s. CLOSE

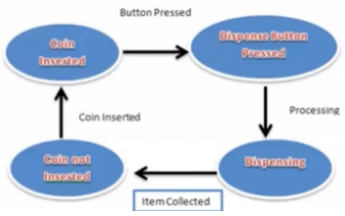
Park vehicle
Get available counts
Free spot

以上use case，的确受Open/Close的影响
但是以上的use case，并不会导致State的转换

State

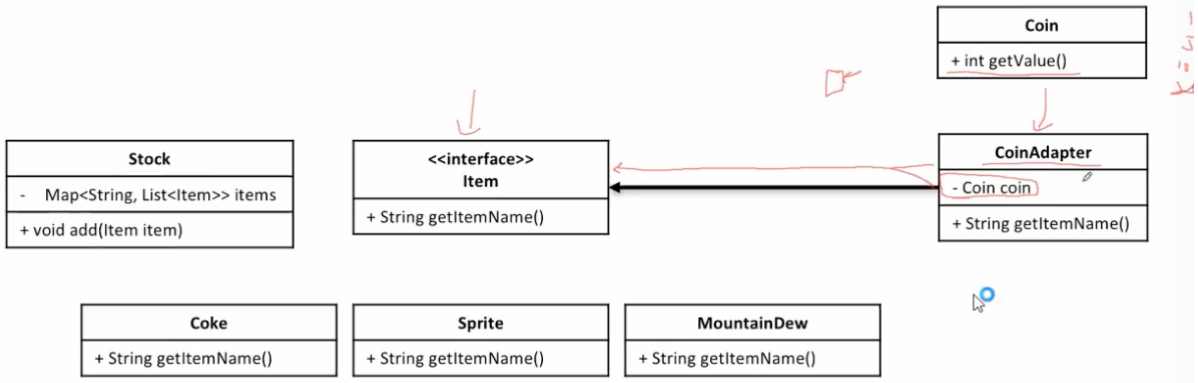
- 出现频率不高
- 特别适合于特殊类型的题目

e.g. 实物类 -> Vending Machine
http://ydtech.blogspot.com/2010/06/state-design-pattern-by-example.html



Adapter

- 例子：



Adapter


```
public class CoinAdapter implements Item
{
    private Coin coin;

    public CoinAdapter(Coin coin)
    {
        this.coin = coin;
    }

    public String getItemName()
    {
        return new String(coin.getValue());
    }
}
```

Strategy v.s. Factory

42

Strategy is about behavior. Factory is about creation/instatation.

Suppose you have an algorithm, to calculate a discount percentage. You can have 2 implementations of that algorithm; one for regular customers, and one for extra-ordinary good customers.
You can use a strategy DP for this implementation: you create an interface, and 2 classes that implement that interface. In one class, you implement the regular discount-calculation algorithm, in the other class you implement the 'good customers' algorithm.

Then, you can use a factory pattern to instantiate the class that you want. The factory method thus instantiates either the regular customer-discount algorithm, or the other implementation.

In short: the factory method instantiates the correct class; the strategy implementation contains the algorithm that must be executed.

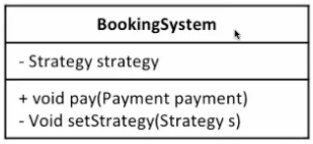
share improve this answer

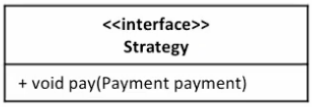
answered Mar 21 '11 at 8:16

Frederik Gheysels

45.8k ● 8 ● 78 ● 136

Strategy v.s. Factory






```
String account = payment.getAccount();
String password = payment.getPassword();
```

```
String cardId = payment.getCardId();
String name = payment.getName();
String cvv = payment.getCvv();
```

Strategy v.s. Factory

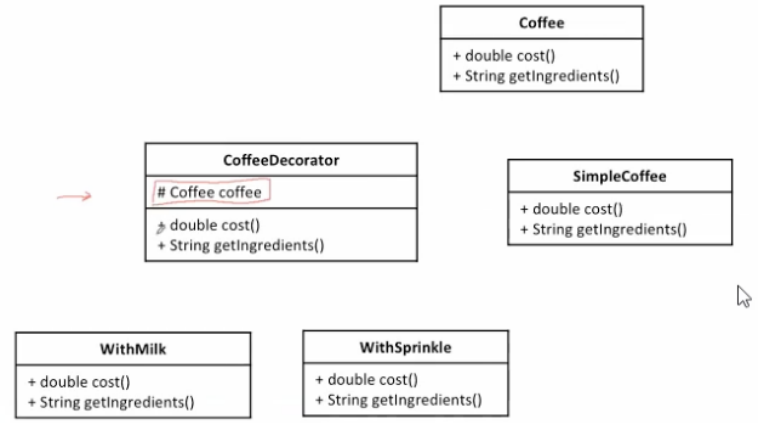
```
public class StrategyFactory
{
    public Strategy createStrategy(Payment payment)
    {
        if(payment.getMethod().equals("paypal"))
        {
            strategy = new PaypalStrategy();
        }
        else if(payment.getMethod().equals("credit card"))
        {
            strategy = new CreditCardStrategy();
        }
    }
}

public void pay(Payment payment)
{
    strategy = createStrategy(payment);
    strategy.processPayment(payment);
}
```

```
public interface Strategy
{
    public void processPayment(Payment payment);
}

public class PaypalStrategy implements Strategy
{
    public void processPayment(Payment payment)
    {
        // get paypal account
        // get paypal password
        // ...
    }
}
```

Decorator



Decorator

```
// Abstract decorator class - note that it implem
public abstract class CoffeeDecorator implements
    protected final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee c) {
        this.decoratedCoffee = c;
    }

    public double getCost() { // Implementing met
        return decoratedCoffee.getCost();
    }

    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}
```

Decorator

九章算法

```java
// The interface Coffee defines the functionality of Coffee implemented by decorator
public interface Coffee {
    public double getCost(); // Returns the cost of the coffee
    public String getIngredients(); // Returns the ingredients of the coffee
}

// Extension of a simple coffee without any extra ingredients
public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1;
    }

    @Override
    public String getIngredients() {
        return "Coffee";
    }
}
```

```java
public class Main {
    public static void printInfo(Coffee c) {
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());
    }

    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSprinkles(c);
        printInfo(c);
    }
}
```

The output of this program is given below:

```
Cost: 1.0; Ingredients: Coffee
Cost: 1.5; Ingredients: Coffee, Milk
Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles
```