0 - Design Connect Four

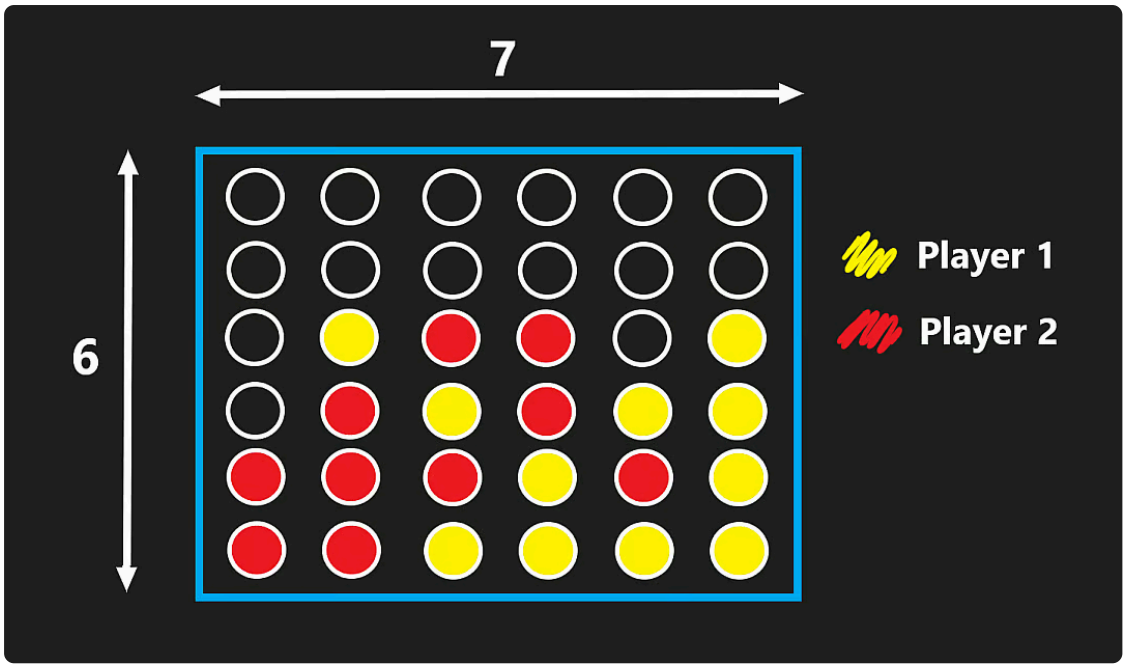
16:46

Mark Lesson Complete View Code

Design Connect Four

Background

Connect Four is a popular game played on a 7x6 grid. Two players take turns dropping colored discs into the grid. The first player to get four discs in a row (vertically, horizontally or diagonally) wins.



Requirements

Some possible questions to ask:

- What are the rules of the game?
- What size is the grid?
- How many players are there? Player vs Computer? Player vs Player?
- Are we keeping track of the score?

Basics

- The game will be played by only two players, player vs player
- The game board should be of variable dimensions
- The target is to connect N discs in a row (vertically, horizontally or diagonally)
    - N is a variable (e.g. connect 4, 5, 6, etc)
- There should be a score tracking system
    - After a player reaches the target score, they are the winner

Design

High-level

- We will need a **Grid** class to maintain the state of the 2-D board

    - The board cell can be empty, yellow (occupied by Player 1) or red (occupied by Player 2)
    - The grid will also be responsible for checking for a win condition
- We can have a **Player** class to represent the player's piece color
    - This isn't super important, but encapsulating information is generally a good practice

- The **Game** class will be composed of the **Grid** and **Players**
    - The **Game** class will be responsible for the game loop and keeping track of the score

Code

We will use an enum to represent the **GridPosition**.

Python JavaScript Java C++

```
import enum

class GridPosition(enum.Enum):
    EMPTY = 0
    YELLOW = 1
    RED = 2
```

The **Grid** will maintain the state of the board and all of the pieces. It will also check for a win condition. Perhaps it would be more appropriate to name the `checkWin` method to `checkNConnected`, since the **Grid** itself shouldn't need to know what the rules of the game are.

Python JavaScript Java C++

```
class Grid:
    def __init__(self, rows, columns):
        self._rows = rows
        self._columns = columns
```

```
        self._grid = None
        self.initGrid()

    def initGrid(self):
        self._grid = [[GridPosition.EMPTY for _ in
range(self._columns)] for _ in range(self._rows)]

    def getGrid(self):
        return self._grid

    def getColumnCount(self):
        return self._columns

    def placePiece(self, column, piece):
        if column < 0 or column >= self._columns:
            raise ValueError('Invalid column')
        if piece == GridPosition.EMPTY:
            raise ValueError('Invalid piece')
        for row in range(self._rows-1, -1, -1):
            if self._grid[row][column] ==
GridPosition.EMPTY:
                self._grid[row][column] = piece
                return row

    def checkWin(self, connectN, row, col, piece):
        count = 0
        # Check horizontal
        for c in range(self._columns):
            if self._grid[row][c] == piece:
                count += 1
            else:
                count = 0
            if count == connectN:
                return True

        # Check vertical
        count = 0
        for r in range(self._rows):
            if self._grid[r][col] == piece:
                count += 1
            else:
                count = 0
            if count == connectN:
```

```
                return True

        # Check diagonal
        count = 0
        for r in range(self._rows):
            c = row + col - r
            if c >= 0 and c < self._columns and
self._grid[r][c] == piece:
                count += 1
            else:
                count = 0
            if count == connectN:
                return True

        # Check anti-diagonal
        count = 0
        for r in range(self._rows):
            c = col - row + r
            if c >= 0 and c < self._columns and
self._grid[r][c] == piece:
                count += 1
            else:
                count = 0
            if count == connectN:
                return True

        return False
```

A **Player** is only meant to encapsulate the player's information, more importantly the player's piece color.

Python JavaScript Java C++

```
class Player:
    def __init__(self, name, pieceColor):
        self._name = name
        self._pieceColor = pieceColor

    def getName(self):
        return self._name
```

```
    def getPieceColor(self):
        return self._pieceColor
```

The **Game** class will be used to play the game. It will keep track of the players, the score, and the grid. It will also be responsible for the game loop. The game parameters passed in via the constructor give us flexibility to play the game with slightly different rules and dimensions.

While we could instantiate the board within the **Game** class, it's preferred to pass it in via the constructor. This means the **Game** class does not need to know how to instantiate the board.

Even though we are only playing with two players, we can still use a list to store the players. This is not necessary, but it's easy enough and gives us flexibility to add more players in the future.

Python JavaScript Java C++

```
class Game:
    def __init__(self, grid, connectN, targetScore):
        self._grid = grid
        self._connectN = connectN
        self._targetScore = targetScore

        self._players = [
            Player('Player 1', GridPosition.YELLOW),
            Player('Player 2', GridPosition.RED)
        ]

        self._score = {}
        for player in self._players:
            self._score[player.getName()] = 0

    def printBoard(self):
        print('Board:\n')
        grid = self._grid.getGrid()
        for i in range(len(grid)):
            row = ''
            for piece in grid[i]:
```

```
                if piece == GridPosition.EMPTY:
                    row += '0 '
                elif piece == GridPosition.YELLOW:
                    row += 'Y '
                elif piece == GridPosition.RED:
                    row += 'R '
            print(row)
        print('')

    def playMove(self, player):
        self.printBoard()
        print(f"{player.getName()}'s turn")
        colCnt = self._grid.getColumnCount()
        moveColumn = int(input(f"Enter column between
{0} and {colCnt - 1} to add piece: "))
        moveRow = self._grid.placePiece(moveColumn,
player.getPieceColor())
        return (moveRow, moveColumn)

    def playRound(self):
        while True:
            for player in self._players:
                row, col = self.playMove(player)
                pieceColor = player.getPieceColor()
                if self._grid.checkWin(self._connectN,
row, col, pieceColor):
                    self._score[player.getName()] += 1
                    return player

    def play(self):
        maxScore = 0
        winner = None
        while maxScore < self._targetScore:
            winner = self.playRound()
            print(f"{winner.getName()} won the round")
            maxScore =
max(self._score[winner.getName()], maxScore)

            self._grid.initGrid() # reset grid
        print(f"{winner.getName()} won the game")
```

Finally, we can create the grid, set the game parameters, and play the game.

Python JavaScript Java C++

```
grid = Grid(6, 7)
game = Game(grid, 4, 2)
game.play()
```

Object-Oriented Design Interview

0 / 7

Interview Questions

1 - Design Blackjack

18:35

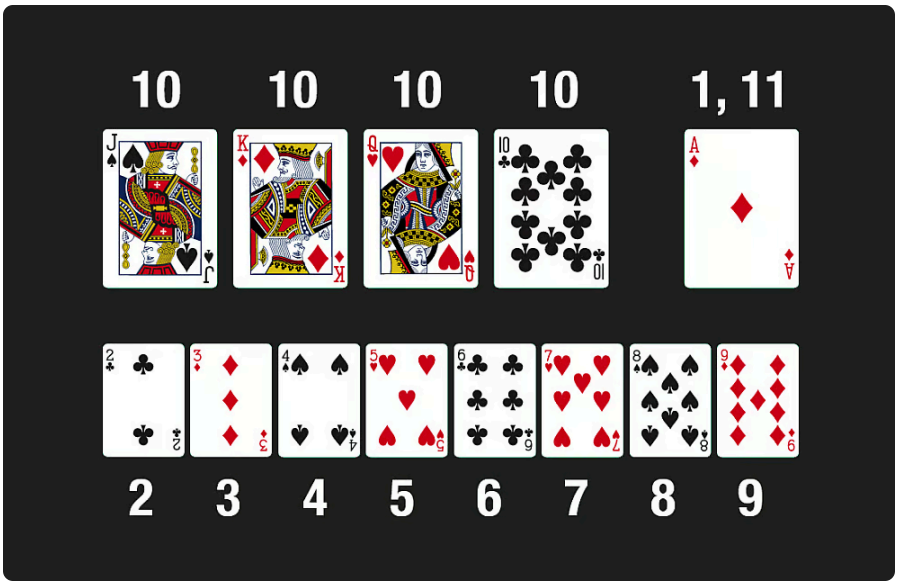
Mark Lesson Complete View Code

Design Blackjack

Background

Blackjack is a popular card game played in casinos. The goal is to get a hand with a value as close to 21 as possible without going over.

The player is dealt (or draws) two cards and can choose to draw more or stop. The dealer is dealt two cards as well, but only one is visible to the player. The player wins if their hand is closer to 21 than the dealer's hand. But if they go over 21 they automatically lose. If the player and dealer have the same value, it's a tie.



Requirements

Obviously this isn't a test of how well you know blackjack. You want to clarify the requirements with the interviewer, because they might have completely different requirements in mind.

Some possible questions to ask:

- How many cards are in a deck? what kind? etc.
- Is the deck refilled after each round?
- How many players?
- Are we implementing gambling or score-keeping?

Basics

- Only two players, including the dealer
- Only one deck of cards, which is refilled after each round
- We are implementing a gambling system for the non-dealer player

Cards

- There are 52 cards in a deck
- Each card has a suit (hearts, spades, diamonds, clubs) and there are 13 cards in each suit
- A card could be
    - numbered (2-10) and have the same point value
    - a face (jack, queen, king) and have a point value of 10
    - an ace and have a point value of 1 or 11, whichever is better for the player

Game Round

- To start each round, both the dealer and the player are dealt two cards
- One of the dealer's cards is hidden from the player
- The player can choose to draw one more card until they go over 21 or decide to stop
    - If they go over 21, they lose and the round is over
- The dealer will draw until they have a hand value of `N` or more, where `N` is the player's hand value
    - If the dealer goes over 21, the player wins
- At the end, if the dealer's hand value is greater than the player's, the dealer wins

Gambling

- A player can start with an arbitrary amount of money

- The player can bet as much money as they have
- The dealer will never run out of money

Design

High-level

- A **Card** will have a **Suit** and a **Value**
    - The suit is irrelevant for blackjack, but if our game had a UI it would be useful information.
    - Similarly, a ten, jack, queen, and king all have the same value, so we won't distinguish between them.
    - But these points would be worth clarifying. You don't want to overengineer a solution, but you also don't want to make major assumptions without clarifying with your interviewer.
- A **Deck** will have an array of **Cards**
    - The deck will be responsible for shuffling and popping (drawing) **Cards**
- A **Player** could be either the **Dealer** or the **User**, and since both will a **Hand**, the **Player** can be an abstract class
    - The **Player** will also be responsible for making a move (drawing or stopping)
- The **UserPlayer** will also have a **Balance** and be able to place a **Bet**

Code

A suit can have 4 discrete values so it makes sense to use an `Enum` for it.

Python	JavaScript	Java	C++

```
from enum import Enum

class Suit(Enum):
    CLUBS, DIAMONDS, HEARTS, SPADES =
'clubs', 'diamonds', 'hearts', 'spades'
```

A card will have a suit and a value. The value will be a number between 1 and 10. We could add logic which enforces the value is valid and throws an exception if it's not, but for simplicity we won't.

Python JavaScript Java C++

```
class Card:
    def __init__(self, suit, value):
        self._suit = suit
        self._value = value

    def getSuit(self):
        return self._suit

    def getValue(self):
        return self._value

    def print(self):
        print(self.getSuit(),
self.getValue())
```

A hand will contain an array of cards, and the total score, which will be the sum of the values of the cards. Notice that we recalculate the score whenever we add a card. We also added logic to handle the case where the card is an ace.

Based on the **Single Responsibility** principle, we could have a separate class which handles printing the cards, but for simplicity we won't.

Python JavaScript Java C++

```
class Hand:
    def __init__(self):
        self._score = 0
        self._cards = []

    def addCard(self, card):
        self._cards.append(card)
        if card.getValue() == 1:
            self._score += 11 if
self._score + 11 <= 21 else 1
        else:
            self._score += card.getValue()
        print('Score: ', self._score)

    def getScore(self):
        return self._score

    def getCards(self):
        return self._cards

    def print(self):
        for card in self.getCards():
            print(card.getSuit(),
card.getValue())
```

The deck will have an array of cards, and will be responsible for shuffling and drawing cards.
The simplest way to build the deck is to iterate 13 times for each suit.

Notice that this way, we can't distinguish between a ten, jack, queen, and king, but for out design we don't need to anyway.

Alternatively, we could have a separate class for each card, but that would be overkill for this problem. A middleground would be to add a `name` field to the card class.

Python JavaScript Java C++

```
import random

class Deck:
    def __init__(self):
        self._cards = []
        for suit in Suit:
            for value in range(1, 14):

self._cards.append(Card(suit, min(value,
10)))

    def print(self):
        for card in self._cards:
            card.print()

    def draw(self):
        return self._cards.pop()

    def shuffle(self):
        for i in range(len(self._cards)):
            j = random.randint(0, 51)
            self._cards[i], self._cards[j]
= self._cards[j], self._cards[i]
```

We also define a **Player** abstract class which has a **Hand** and can make a move. The **Dealer** and **UserPlayer** will extend this class to override `makeMove()`.

Python JavaScript Java C++

```python
from abc import ABC, abstractmethod

class Player(ABC):
    def __init__(self, hand):
        self._hand = hand

    def getHand(self):
        return self._hand

    def clearHand(self):
        self._hand = Hand()

    def addCard(self, card):
        self._hand.addCard(card)

    @abstractmethod
    def makeMove(self):
        pass
```

The **UserPlayer** will have a **Balance** and be able to place a **Bet**. It will also override `makeMove()` to prompt the user for input: returning `true` to draw a card and `false` to stop.
Alternatively, we could have implemented receiving user input from another class, but I wanted to illustrate that the player is responsible for making a move.

Python JavaScript Java C++

```python
class UserPlayer(Player):
    def __init__(self, balance, hand):
        super().__init__(hand)
        self._balance = balance

    def getBalance(self):
        return self._balance

    def placeBet(self, amount):
        if amount > self._balance:
            raise ValueError('Insufficient
funds')
        self._balance -= amount
        return amount

    def receiveWinnings(self, amount):
        self._balance += amount

    def makeMove(self):
        if self.getHand().getScore() > 21:
            return False
        move = input('Draw card? [y/n] ')
        return move == 'y'
```

The **Dealer** will be less involved since they don't need to place bets or receive winnings. It will also override `makeMove()` but they will draw until their hand value is greater than or equal to some `targetScore`.
In the our **GameRound** this `targetScore` will be the player's hand value.
The reason we can't pass the `targetScore` in the constructor is because the dealer will need to update it after the player draws.

Python JavaScript Java C++

```
class Dealer(Player):
    def __init__(self, hand):
        super().__init__(hand)
        self._targetScore = 17

    def updateTargetScore(self, score):
        self._targetScore = score

    def makeMove(self):
        return self.getHand().getScore() <
self._targetScore
```

The **GameRound** will be responsible for controlling the flow of the game. It must be provided a **UserPlayer**, **Dealer**, and **Deck**.

It will also be responsible for prompting the user for a bet amount, dealing the initial cards, and cleaning up the round.

We added plenty of print statements for clarity, but in a real interview you could be more concise.

Python JavaScript Java C++

```
class GameRound:
    def __init__(self, player, dealer,
deck):
        self._player = player
        self._dealer = dealer
        self._deck = deck

    def getBetUser(self):
        amount = int(input('Enter a bet
amount: '))
```

```
        return amount

    def dealInitialCards(self):
        for i in range(2):

self._player.addCard(self._deck.draw())

self._dealer.addCard(self._deck.draw())
        print('Player hand: ')
        self._player.getHand().print()
        dealerCard =
self._dealer.getHand().getCards()[0]
        print("Dealer's first card: ")
        dealerCard.print()

    def cleanupRound(self):
        self._player.clearHand()
        self._dealer.clearHand()
        print('Player balance: ',
self._player.getBalance())

    def play(self):
        self._deck.shuffle()

        if self._player.getBalance() <= 0:
            print('Player has no more money
=)')
            return
        userBet = self.getBetUser()
        self._player.placeBet(userBet)

        self.dealInitialCards()

        # User makes moves
        while self._player.makeMove():
            drawnCard = self._deck.draw()
            print('Player draws',
drawnCard.getSuit(), drawnCard.getValue())
            self._player.addCard(drawnCard)
            print('Player score: ',
```

```
self._player.getHand().getScore())

        if
self._player.getHand().getScore() > 21:
            print('Player busts!')
            self.cleanupRound()
            return

        # Dealer makes moves
        while self._dealer.makeMove():

self._dealer.addCard(self._deck.draw())

        # Determine winner
        if
self._dealer.getHand().getScore() > 21 or
self._player.getHand().getScore() >
self._dealer.getHand().getScore():
            print('Player wins')

self._player.receiveWinnings(userBet * 2)
        elif
self._dealer.getHand().getScore() >
self._player.getHand().getScore():
            print('Player loses')
        else:
            print('Game ends in a draw')

self._player.receiveWinnings(userBet)
        self.cleanupRound()
```

Finally, we can run the game until the player runs out of money.

Python JavaScript Java C++

```
player = UserPlayer(1000, Hand())
dealer = Dealer(Hand())
```

```
while player.getBalance() > 0:
    gameRound = GameRound(player, dealer,
Deck()).play()
```

Object-Oriented Design Interview

0 / 7

Interview Questions

2 - Design a Parking Lot

16:31

Mark Lesson Complete View Code

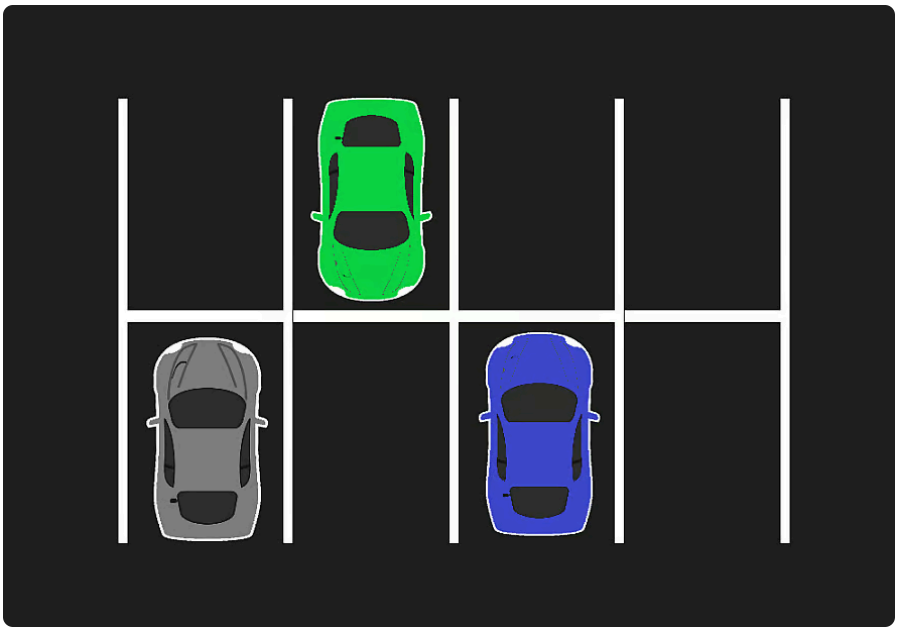
Design a Parking Lot

Background

Parking lots have open spaces for vehicles to park in. Vehicles can be of different sizes, e.g. cars, limos, trucks, etc.

In some cases, parking spots can be numbered. For large venues, parking lots may have multiple levels, i.e. parking garages.

Sometimes parking is free, but in other cases customers have to pay. So parking lots can have a payment system to keep track of parked vehicles.



Requirements

Some possible questions to ask:

- Will there be multiple levels in the parking lot?
- What kinds of vehicles will be parked? Will their sizes differ?
- Will there be special spots for certain vehicles?
- Will the parking lot have a payment system? If so, how will it work?
- Will parking spots be reserved or can the driver choose any spot?
- How much functionality will the driver have beyond parking and paying?

Basics

- Multiple levels in the parking lot
- Possible vehicle types: car, limo, semi-truck
- We will have a payment system, with a single entrance and exit
- Drivers will be assigned a parking spot after paying

Vehicles and Parking Spots

- Vehicles can be of different sizes (car = 1, limo = 2, truck = 3)
- Each parking spot will have a size of 1
    - A vehicle must fully take up each spot assigned to it (no fractional spots)
- Vehicles will automatically be assigned the next available parking spot on the lowest floor

Payment System

- Drivers will pay for parking and be assigned the next available spot on the lowest floor
- Drivers can pay for a variable number of hours and they are charged after they remove their vehicle based on an hourly rate
    - We can assume vehicles can be parked for a variable number of hours
- If there is no capacity, the system should not assign a spot and should notify the driver

Design

High-level

- We will have a base **Vehicle** class, and **Car**, **Limo**, and **Truck** classes that inherit from it
- Each of these will have a predefined size
- A **Driver** class will have a vehicle that belongs to it, and a total payment due
- We will have a **ParkingGarage** which will be made up of multiple **ParkingFloors**
- A **ParkingFloor** will be made up of multiple Parking Spots, which can simply be represented by an array (`0` = empty, `1` = occupied)
- The **ParkingSystem** will be the main controller of the **ParkingGarage** and will be responsible for tracking parking hours and charging drivers

Code

A **Vehicle** class will be the base class for all vehicles. It will have a size attribute that will be used to determine how many spots it will take up.

Python JavaScript Java C++

```python
class Vehicle:
    def __init__(self, spot_size):
        self._spot_size = spot_size

    def get_spot_size(self):
        return self._spot_size
```

A **Driver** class will have a vehicle that belongs to it, and a total payment due. It will also have a method to charge the driver for parking.

Python JavaScript Java C++

```python
class Driver:
    def __init__(self, id, vehicle):
        self._id = id
        self._vehicle = vehicle
        self._payment_due = 0

    def get_vehicle(self):
        return self._vehicle

    def get_id(self):
        return self._id

    def charge(self, amount):
        self._payment_due += amount
```

Cars, **Limos**, and **SemiTrucks** will inherit from the **Vehicle** class. Each will have a predefined size.

Python JavaScript Java C++

```python
class Car(Vehicle):
    def __init__(self):
        super().__init__(1)

class Limo(Vehicle):
    def __init__(self):
        super().__init__(2)

class SemiTruck(Vehicle):
    def __init__(self):
        super().__init__(3)
```

A **ParkingFloor** will be the container for parking spots, which will be represented by an array. It will also keep track of which vehicles are parked in which spots, where `[l, r]` represents the range of spots occupied by a vehicle.

Python JavaScript Java C++

```
class ParkingFloor:
    def __init__(self, spot_count):
        self._spots = [0] * spot_count
        self._vehicle_map = {}

    def park_vehicle(self, vehicle):
        size = vehicle.get_spot_size()
        l, r = 0, 0
        while r < len(self._spots):
            if self._spots[r] != 0:
                l = r + 1
            if r - l + 1 == size:
                # we found enough spots,
park the vehicle
                for k in range(l, r+1):
                    self._spots[k] = 1
                self._vehicle_map[vehicle]
= [l, r]
                return True
            r += 1
        return False

    def remove_vehicle(self, vehicle):
        start, end =
self._vehicle_map[vehicle]
        for i in range(start, end + 1):
            self._spots[i] = 0
        del self._vehicle_map[vehicle]

    def get_parking_spots(self):
```

```
        return self._spots

    def get_vehicle_spots(self, vehicle):
        return
self._vehicle_map.get(vehicle)
```

A **ParkingGarage** will contain an arbitrary number of **ParkingFloors**.

Notice how `spots_per_floor` is passed into the **ParkingFloor** constructor. This is because we want to be able to have different sized parking floors. But what if each floor had a varying number of parking spots? We could pass in an array of `spots_per_floor` instead.

Python **JavaScript** **Java** **C++**

```
class ParkingGarage:
    def __init__(self, floor_count,
spots_per_floor):
        self._parking_floors =
[ParkingFloor(spots_per_floor) for _ in
range(floor_count)]

    def park_vehicle(self, vehicle):
        for floor in self._parking_floors:
            if floor.park_vehicle(vehicle):
                return True
        return False

    def remove_vehicle(self, vehicle):
        for floor in self._parking_floors:
            if
floor.get_vehicle_spots(vehicle):

floor.remove_vehicle(vehicle)
```

```
                return True
        return False
```

The **ParkingSystem** will be the main controller of the **ParkingGarage**. It will be responsible for tracking parking hours and charging drivers.

Python JavaScript Java C++

```
import datetime
import math

class ParkingSystem:
    def __init__(self, parkingGarage,
hourlyRate):
        self._parkingGarage = parkingGarage
        self._hourlyRate = hourlyRate
        self._timeParked = {} # map
driverId to time that they parked

    def park_vehicle(self, driver):
        currentHour =
datetime.datetime.now().hour
        isParked =
self._parkingGarage.park_vehicle(driver.get
_vehicle())
        if isParked:

self._timeParked[driver.get_id()] =
currentHour
        return isParked

    def remove_vehicle(self, driver):
        if driver.get_id() not in
self._timeParked:
            return False
        currentHour =
datetime.datetime.now().hour
```

```
        timeParked = math.ceil(currentHour
- self._timeParked[driver.get_id()])
        driver.charge(timeParked *
self._hourlyRate)

        del
self._timeParked[driver.get_id()]
        return
self._parkingGarage.remove_vehicle(driver.g
et_vehicle())
```

Finally, we can test it out by parking a few vehicles.

Python JavaScript Java C++

```
parkingGarage = ParkingGarage(3, 2)
parkingSystem =
ParkingSystem(parkingGarage, 5)

driver1 = Driver(1, Car())
driver2 = Driver(2, Limo())
driver3 = Driver(3, SemiTruck())

print(parkingSystem.park_vehicle(driver1))
# true
print(parkingSystem.park_vehicle(driver2))
# true
print(parkingSystem.park_vehicle(driver3))
# false

print(parkingSystem.remove_vehicle(driver1)
)    # true
print(parkingSystem.remove_vehicle(driver2)
)    # true
print(parkingSystem.remove_vehicle(driver3)
)    # false
```

3 - Design a Bank

15:23

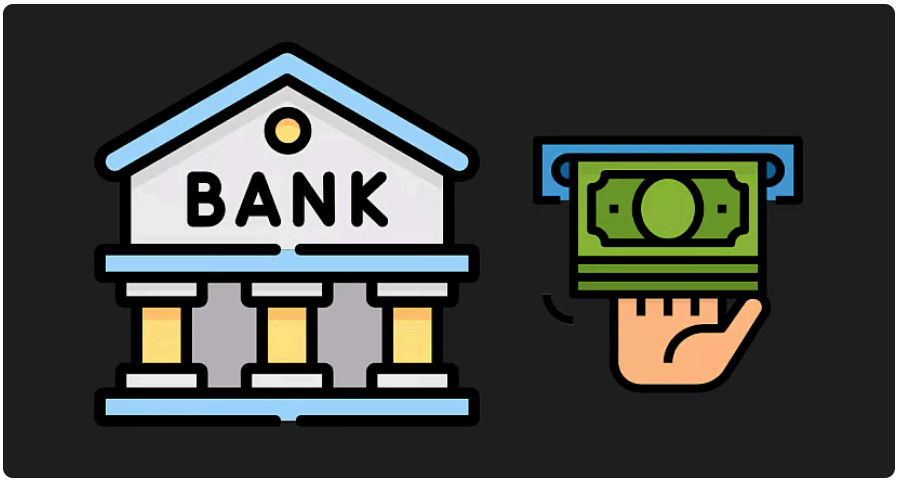
Mark Lesson Complete View Code

Design a Bank

Background

Banks offer a wide variety of financial services, including checking and savings accounts, credit cards, and loans. Customers typically have to open an account with a bank prior to using any services. Customers can

deposit or withdraw money, and even purchase investments.



Requirements

Some possible questions to ask:

- What financial services will the bank offer?
- Will customers be required to have accounts? Will the bank manage them?
- Will the bank have physical locations and bank tellers?
- Are we concerned with physical security of the bank? i.e. will the bank have a vault?

Services

- Customers can open accounts and deposit/withdraw money
- We are only concerned with transactions that take place within a physical location (i.e. through a bank teller)

Tellers

- Tellers can perform transactions on behalf of customers
    - Every transaction is recorded and associated with the Teller and Customer

Headquarters

- Each branch location will send money to a central location (i.e. the bank's headquarters) at the end of the day
    - We don't need to worry about the transporation details

Design

High-level

- We will have a base **Transaction** class that will be inherited by **Deposit**, **Withdrawal**, and **OpenAccount** classes.
- A **BankTeller** will simply encapsulate the unique ID of the teller. We don't need a class for Customer's, since we can use the **BankAccount** class to encapsulate their **ID** and **Balance**.
- A headquarter **Bank** will be made up of multiple **BankBranch** objects, and a single **BankSystem** which will be the central store for customer accounts and transactions.
    - Note that a customer could transact with multiple branches, so we need to store their information in the **BankSystem**.

Code

A **Transaction** will be tied to a customer and a teller. We will have a **get_transaction_description** method that will be implemented by the child classes.
This design follows the **Open-Closed Principle** since we can add new transaction types without modifying the **Transaction** class.

But also provides more flexibility than using an `enum` since we can encapsulate additional information.

Python JavaScript Java C++

```
from abc import ABC, abstractmethod

class Transaction(ABC):
    def __init__(self, customerId,
tellerId):
        self._customerId = customerId
        self._tellerId = tellerId

    def get_customer_id(self):
        return self._customerId

    def get_teller_id(self):
        return self._tellerId

    @abstractmethod
    def get_transaction_description(self):
        pass
```

The three transaction types will inherit from **Transaction** and implement the **get_transaction_description** method.

Python JavaScript Java C++

```
class Deposit(Transaction):
    def __init__(self, customerId,
tellerId, amount):
        super().__init__(customerId,
tellerId)
        self._amount = amount
```

```
    def get_transaction_description(self):
        return f'Teller
{self.get_teller_id()} deposited
{self._amount} to account
{self.get_customer_id()}'

class Withdrawal(Transaction):
    def __init__(self, customerId,
tellerId, amount):
        super().__init__(customerId,
tellerId)
        self._amount = amount

    def get_transaction_description(self):
        return f'Teller
{self.get_teller_id()} withdrew
{self._amount} from account
{self.get_customer_id()}'

class OpenAccount(Transaction):
    def __init__(self, customerId,
tellerId):
        super().__init__(customerId,
tellerId)

    def get_transaction_description(self):
        return f'Teller
{self.get_teller_id()} opened account
{self.get_customer_id()}'
```

Python JavaScript Java C++

```
class BankTeller:
    def __init__(self, id):
        self._id = id
```

```
    def get_id(self):
        return self._id
```

A **BankAccount** encapsulates a customer's information, along with their balance.

Python JavaScript Java C++

```
class BankAccount:
    def __init__(self, customerId, name,
balance):
        self._customerId = customerId
        self._name = name
        self._balance = balance

    def get_balance(self):
        return self._balance

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount
```

A **BankSystem** is the central store for all customer accounts and transaction logs, regardless of which **BankBranch** they take place at.
For simplicity, we store new accounts in an array, where the ID of the customer account will be the next available index in the array.

Python JavaScript Java C++

```
class BankSystem:
    def __init__(self, accounts,
transactions):
        self._accounts = accounts
        self._transactions = transactions

    def get_account(self, customerId):
        return self._accounts[customerId]

    def get_accounts(self):
        return self._accounts

    def get_transactions(self):
        return self._transactions

    def open_account(self, customer_name,
teller_id):
        # Create account
        customerId =
len(self.get_accounts())
        account = BankAccount(customerId,
customer_name, 0)
        self._accounts.append(account)

        # Log transaction
        transaction =
OpenAccount(customerId, teller_id)

self._transactions.append(transaction)
        return customerId

    def deposit(self, customer_id,
teller_id, amount):
        account =
self.get_account(customer_id)
        account.deposit(amount)

        transaction = Deposit(customer_id,
teller_id, amount)
```

```
self._transactions.append(transaction)

    def withdraw(self, customer_id,
teller_id, amount):
        if amount >
self.get_account(customer_id).get_balance()
:
            raise Exception('Insufficient
funds')
        account =
self.get_account(customer_id)
        account.withdraw(amount)

        transaction =
Withdrawal(customer_id, teller_id, amount)

self._transactions.append(transaction)
```

A **BankBranch** will be responsible for performing transactions on behalf of customers via available **BankTellers**.

We also add methods for cash to be collected from and provided to the **BankBranch** (via the headquarter **Bank**).

Python JavaScript Java C++

```
import random

class BankBranch:
    def __init__(self, address,
cash_on_hand, bank_system):
        self._address = address
        self._cash_on_hand = cash_on_hand
        self._bank_system = bank_system
        self._tellers = []
```

```
    def add_teller(self, teller):
        self._tellers.append(teller)

    def _get_available_teller(self):
        index = round(random.random() *
(len(self._tellers) - 1))
        return
self._tellers[index].get_id()

    def open_account(self, customer_name):
        if not self._tellers:
            raise ValueError('Branch does
not have any tellers')
        teller_id =
self._get_available_teller()
        return
self._bank_system.open_account(customer_nam
e, teller_id)

    def deposit(self, customer_id, amount):
        if not self._tellers:
            raise ValueError('Branch does
not have any tellers')
        teller_id =
self._get_available_teller()

self._bank_system.deposit(customer_id,
teller_id, amount)

    def withdraw(self, customer_id,
amount):
        if amount > self._cash_on_hand:
            raise ValueError('Branch does
not have enough cash')
        if not self._tellers:
            raise ValueError('Branch does
not have any tellers')
        self._cash_on_hand -= amount
        teller_id =
```

```
self._get_available_teller()

self._bank_system.withdraw(customer_id,
teller_id, amount)

    def collect_cash(self, ratio):
        cash_to_collect =
round(self._cash_on_hand * ratio)
        self._cash_on_hand -=
cash_to_collect
        return cash_to_collect

    def provide_cash(self, amount):
        self._cash_on_hand += amount
```

The headquarter **Bank** will be responsible for managing all **BankBranches**, as well as collecting cash from each branch.

For convenience, we also add a method to print all transactions that have taken place.

Python JavaScript Java C++

```
class Bank:
    def __init__(self, branches,
bank_system, total_cash):
        self._branches = branches
        self._bank_system = bank_system
        self._total_cash = total_cash

    def add_branch(self, address,
initial_funds):
        branch = BankBranch(address,
initial_funds, self._bank_system)
        self._branches.append(branch)
        return branch
```

```
    def collect_cash(self, ratio):
        for branch in self._branches:
            cash_collected =
branch.collect_cash(ratio)
            self._total_cash +=
cash_collected

    def print_transactions(self):
        for transaction in
self._bank_system.get_transactions():

print(transaction.get_transaction_descripti
on())
```

Finally, we run a simple simulation to demonstrate the functionality of our bank system.

Python | JavaScript | Java | C++

```
bankSystem = BankSystem([], [])
bank = Bank([], bankSystem, 10000)

branch1 = bank.add_branch('123 Main St',
1000)
branch2 = bank.add_branch('456 Elm St',
1000)

branch1.add_teller(BankTeller(1))
branch1.add_teller(BankTeller(2))
branch2.add_teller(BankTeller(3))
branch2.add_teller(BankTeller(4))

customerId1 = branch1.open_account('John
Doe')
customerId2 = branch1.open_account('Bob
Smith')
customerId3 = branch2.open_account('Jane
Doe')
```

```
branch1.deposit(customerId1, 100)
branch1.deposit(customerId2, 200)
branch2.deposit(customerId3, 300)

branch1.withdraw(customerId1, 50)
""" Possible Output:
    Teller 1 opened account 0
    Teller 2 opened account 1
    Teller 3 opened account 2
    Teller 1 deposited 100 to account 0
    Teller 2 deposited 200 to account 1
    Teller 4 deposited 300 to account 2
    Teller 2 withdrew 50 from account 0
"""

bank.print_transactions()
```

4 - Design a Movie Recommendation System

17:16

Mark Lesson Complete View Code

Design a Movie Recommendation System

Background

Requirements

Some possible questions to ask:

- How will users rate movies?
- How will users be recommended movies?
- What is the expected number of users and movies?
- Edge cases:
    - What if there's a tie in recommended movies, which to recommend?
    - What if a user has not rated any movies?

Basics

- Users can rate movies on a scale of 1 to 5
- Recommend movies based on other users who have rated movies similar to the user's rated movies
    - If there is a tie, recommend any movie from the top movies
- If a user has not rated any movies, recommend the top rated movie

Design

High-level

- We will have a **Movie** class to encapsulate movie information
- And a **User** class to encapsulate user information
- We can use a **MovieRating** enum to represent the rating scale
- To store the ratings of movies by users, we can use a **RatingRegister** class
    - This class will have a map of users to movies they have rated
    - And a map of movies to the ratings they have received
    - We can also store a list of movies and users in the class
- To recommend movies, we can use a **MovieRecommender** class
    - This class will have a reference to the **RatingRegister** class
    - All computations to recommend a movie will be done in this class

Code

The **Movie** class has an `id` to uniquely identify the movie and a `title` to represent the movie's name.

Python | JavaScript | Java | C++

```
class Movie:
    def __init__(self, id, title):
        self._id = id
        self._title = title

    def getId(self):
```

```
        return self._id

    def getTitle(self):
        return self._title
```

The **User** class has an `id` to uniquely identify the user and a `name` which isn't used in our case.

Python JavaScript Java C++

```
class User:
    def __init__(self, id, name):
        self._id = id
        self._name = name

    def getId(self):
        return self._id
```

The **MovieRating** enum represents the rating scale from 1 to 5.

Python JavaScript Java C++

```
from enum import Enum

class MovieRating(Enum):
    NOT_RATED = 0
    ONE = 1
    TWO = 2
    THREE = 3
    FOUR = 4
    FIVE = 5
```

The **RatingRegister** class stores the ratings of movies by users, along with a mapping of users to movies they have rated. These extra mappings are useful to recommend movies to users in an efficient way.

Python JavaScript Java C++

```
class RatingRegister:
    def __init__(self):
        self._userMovies = {}   # Map<UserId, List<Movie>>
        self._movieRatings = {} # Map<MovieId, Map<UserId, Rating>>

        self._movies = []       # List<Movie>
        self._users = []        # List<User>

    def addRating(self, user, movie, rating):
        if movie.getId() not in self._movieRatings:

            self._movieRatings[movie.getId()] = {}
            self._movies.append(movie)
        if user.getId() not in self._userMovies:
            self._userMovies[user.getId()] = []
            self._users.append(user)

        self._userMovies[user.getId()].append(movie)
        self._movieRatings[movie.getId()][user.getId()] = rating

    def getAverageRating(self, movie):
```

```
        if movie.getId() not in
self._movieRatings:
            return
MovieRating.NOT_RATED.value
        ratings =
self._movieRatings[movie.getId()].values()
        ratingValues = [rating.value for
rating in ratings]
        return sum(ratingValues) /
len(ratings)

    def getUsers(self):
        return self._users

    def getMovies(self):
        return self._movies

    def getUserMovies(self, user):
        return
self._userMovies.get(user.getId(), [])

    def getMovieRatings(self, movie):
        return
self._movieRatings.get(movie.getId(), {})
```

The **MovieRecommendation** class has a reference to the **RatingRegister** class. It uses this reference to recommend movies to users.

- If a user has not rated any movies, we recommend the top rated movie
- Otherwise we recommend a movie based on the similarity score of the user with other users
    - We compute the similarity score by comparing the ratings of movies rated by the user and other users
    - We recommend an unwatched movie that has the highest similarity score

Python JavaScript Java C++

```python
class MovieRecommendation:
    def __init__(self, ratings):
        self._movieRatings = ratings

    def recommendMovie(self, user):
        if
len(self._movieRatings.getUserMovies(user))
== 0:
            return
self._recommendMovieNewUser()
        else:
            return
self._recommendMovieExistingUser(user)

    def _recommendMovieNewUser(self):
        best_movie = None
        best_rating = 0
        for movie in
self._movieRatings.getMovies():
            rating =
self._movieRatings.getAverageRating(movie)
            if rating > best_rating:
                best_movie = movie
                best_rating = rating
        return best_movie.getTitle() if
best_movie else None

    def _recommendMovieExistingUser(self,
user):
        best_movie = None
        similarity_score = float('inf') #
Lower is better

        for reviewer in
self._movieRatings.getUsers():
            if reviewer.getId() ==
user.getId():
                continue
```

```
            score =
self._getSimilarityScore(user, reviewer)
            if score < similarity_score:
                similarity_score = score
                recommended_movie =
self._recommendUnwatchedMovie(user,
reviewer)
                best_movie =
recommended_movie if recommended_movie else
best_movie
        return best_movie.getTitle() if
best_movie else None

    def _getSimilarityScore(self, user1,
user2):
        user1_id = user1.getId()
        user2_id = user2.getId()
        user2_movies =
self._movieRatings.getUserMovies(user2)
        score = float('inf') # Lower is
better

        for movie in user2_movies:
            cur_movie_ratings =
self._movieRatings.getMovieRatings(movie)
            if user1_id in
cur_movie_ratings:
                score = 0 if score ==
float('inf') else score
                score +=
abs(cur_movie_ratings[user1_id].value -
cur_movie_ratings[user2_id].value)
        return score

    def _recommendUnwatchedMovie(self,
user, reviewer):
        user_id = user.getId()
        reviewer_id = reviewer.getId()
        best_movie = None
        best_rating = 0
```

```
        reviewer_movies =
self._movieRatings.getUserMovies(reviewer)
        for movie in reviewer_movies:
            cur_movie_ratings =
self._movieRatings.getMovieRatings(movie)
            if user_id not in
cur_movie_ratings and
cur_movie_ratings[reviewer_id].value >
best_rating:
                best_movie = movie
                best_rating =
cur_movie_ratings[reviewer_id].value
        return best_movie
```

An example of how to use the classes:

- User 1 and User 2 have similar tastes, so we recommend them an unwatched movie from each other
- User 3 has not rated any movies, so we recommend them the top rated movie

Python JavaScript Java C++

```
user1 = User(1, 'User 1')
user2 = User(2, 'User 2')
user3 = User(3, 'User 3')

movie1 = Movie(1, 'Batman Begins')
movie2 = Movie(2, 'Liar Liar')
movie3 = Movie(3, 'The Godfather')

ratings = RatingRegister()
ratings.addRating(user1, movie1,
MovieRating.FIVE)
ratings.addRating(user1, movie2,
MovieRating.TWO)
ratings.addRating(user2, movie2,
```

```
MovieRating.TWO)
ratings.addRating(user2, movie3,
MovieRating.FOUR)

recommender = MovieRecommendation(ratings)

print(recommender.recommendMovie(user1)) #
The Godfather
print(recommender.recommendMovie(user2)) #
Batman Begins
print(recommender.recommendMovie(user3)) #
Batman Begins
```

Object-Oriented Design Interview

0 / 7

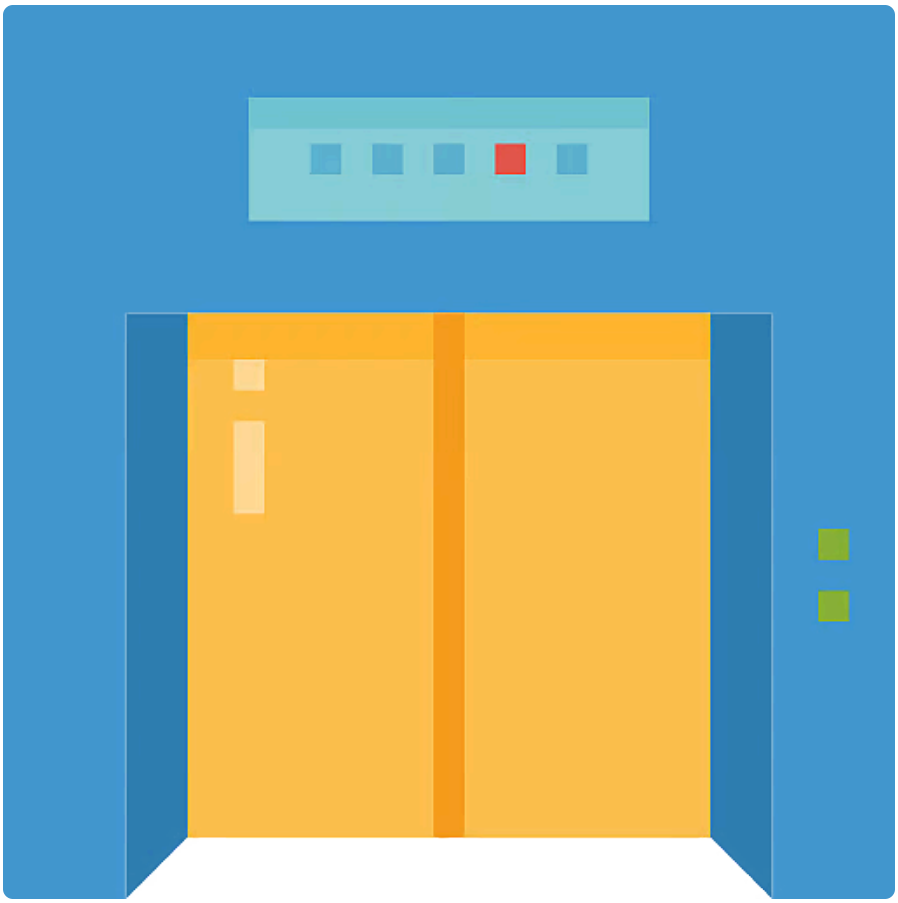
Interview Questions

5 - Design an Elevator System

13:51

Mark Lesson Complete View Code

Design an elevator



Background

An elevator is a mechanical device that moves people or goods between different levels in a building. Elevators provide convenient and efficient vertical transportation, enhancing accessibility, saving time, and maximizing space in tall buildings. They come equipped with safety features to ensure passenger well-being.

The elevator consists of a cabin, a control system, and buttons or controls. These components work in tandem to ensure the elevator operates smoothly and efficiently.

Requirements

Requirements are typically categorized into functional and non-functional. However, before categorizing, it's

essential to gather these requirements. Here are some pivotal questions to present to the interviewer:

1. How many elevators are in the system?
2. How do the elevators process requests? Are requests accepted both internally and externally?
3. How are the requests prioritized? Is it directional or based on proximity?
4. How are intermediate requests addressed?
5. Regarding state management, how does the elevator determine its direction and current activity?
6. Should the elevator initiate in an IDLE state?
7. How is safety ensured within the elevator?
8. How does the elevator maintain efficiency?
9. From a user experience perspective, what feedback does the elevator provide?
10. What measures are in place to emulate the elevator's real-world behaviors?

The above questions will help us gather a comprehensive set of requirements for our elevator system.

Functional Requirements:

Request Handling: The elevator can handle requests from both inside and outside. This applies to both the Passenger elevator and the Service elevator.

Directional Prioritization: Once the elevator starts moving in one direction (either up or down), it will finish processing all requests in that direction before starting

on requests in the opposite direction. The service elevator runs on a first come first serve basis.

Intermediate Stops: While processing requests in one direction, the elevator can stop at intermediate floors to pick up or drop off passengers if there are pending requests.

State Management: The elevator maintains a state (`IDLE`, `GOING_UP`, `GOING_DOWN`) that determines its current activity and direction.

Idle State: If there are no pending requests, the elevator remains in an idle state.

Non-Functional Requirements:

Safety:

The elevator does not change its upward or downward direction suddenly, ensuring it doesn't endanger passengers due to abrupt motion.

Efficiency: Directional prioritization ensures that the elevator does not waste energy by frequently changing directions. It moves in one continuous direction as much as possible.

Scalability: The system is designed in a way that it could potentially handle multiple elevators (though this would need additional coordination logic).

Usability: Clear logs and print statements give the user (or maintenance personnel) a clear idea of the

elevator's activities.

Reliability: The elevator aims to process every request in the queue, assisting all passengers to reach their desired destinations.

Responsive: The elevator responds to requests in a timely manner, based on its prioritization algorithm. The passenger elevator prioritizes the direction and the service elevator prioritizes requests on a first come first serve basis.

Simulated Delays: The system simulates real-world delays like the time taken to travel between floors and the time doors remain open for passengers to embark/disembark.

> In a real world scenario, we would need to consider the timing of the requests, that is, if the elevator were traveling from floor 3 to floor 6, and a request from floor 5 was made, it would not stop abruptly to serve floor 5. However, for the purpose of this implementation, we are assuming that no such requests are made.

Design and Implementation

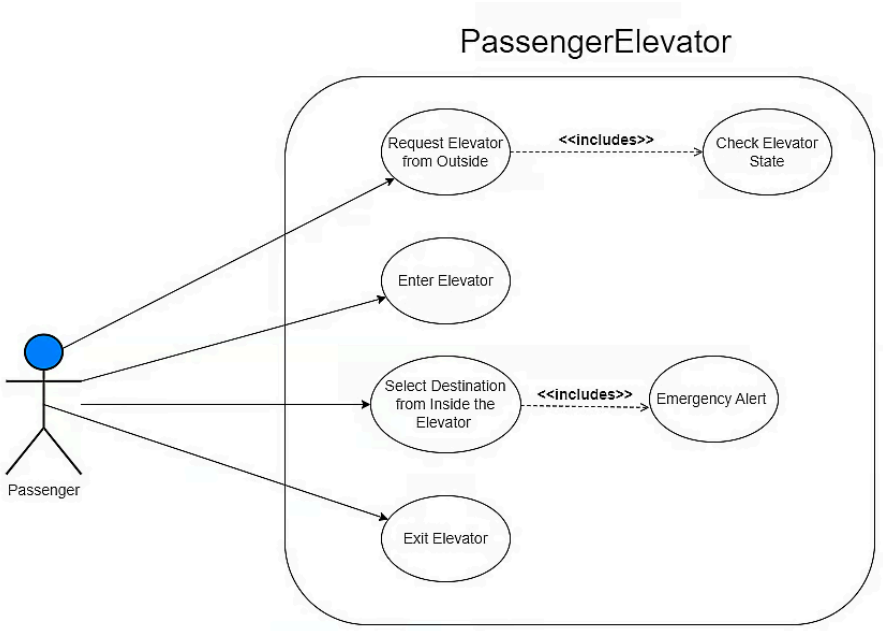
With our functional and non-functional requirements established, we'll now translate these into an object-oriented design (OOD) framework. This involves reinterpreting our requirements, highlighting distinct entities, attributes, behaviors, and relationships

explicitly. The goal is to identify essential classes or objects and understand their interactions.

Design

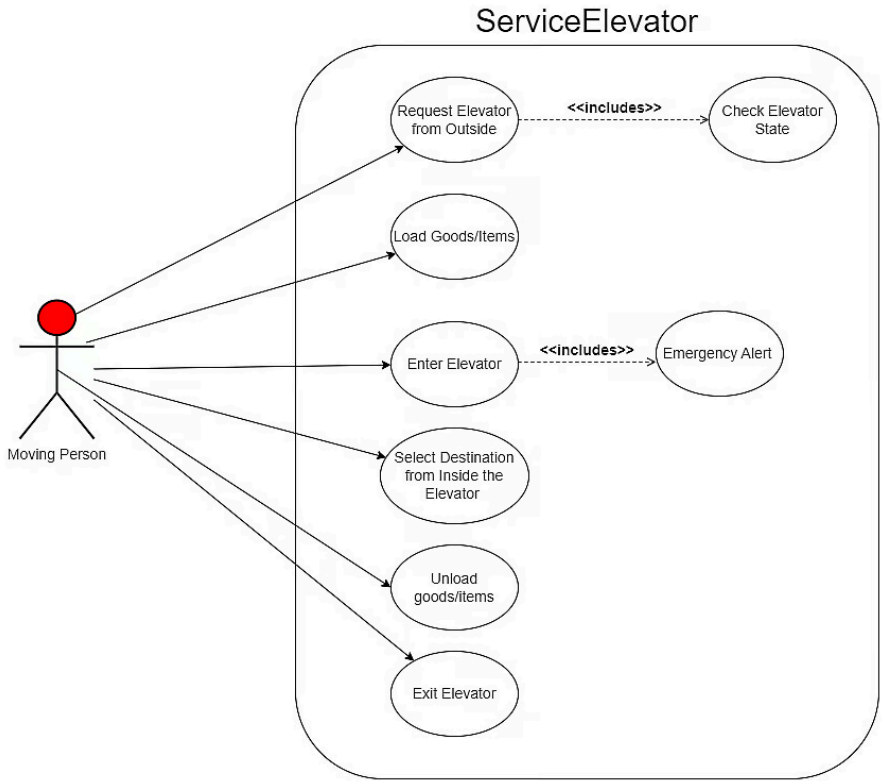
Before delving into code implementation, it's invaluable to outline our use-cases within a use-case diagram and to map out our classes in a UML class diagram. Both of these visual aids will serve as foundational blueprints guiding our coding implementation.

Passenger Elevator Use Cases



> The visual above demonstrates the use-case diagram for the `PassengerElevator`. The passenger interacts by requesting the elevator from outside and by selecting the destination floor inside.

Service Elevator Use Cases



> The visual above demonstrates the use-case diagram for the `ServiceElevator`. The moving person interacts by requesting the elevator, loading/unloading goods, and selecting the destination floor.

UML Class Diagram

The UML class diagram below represents the overall structure of our implementation.

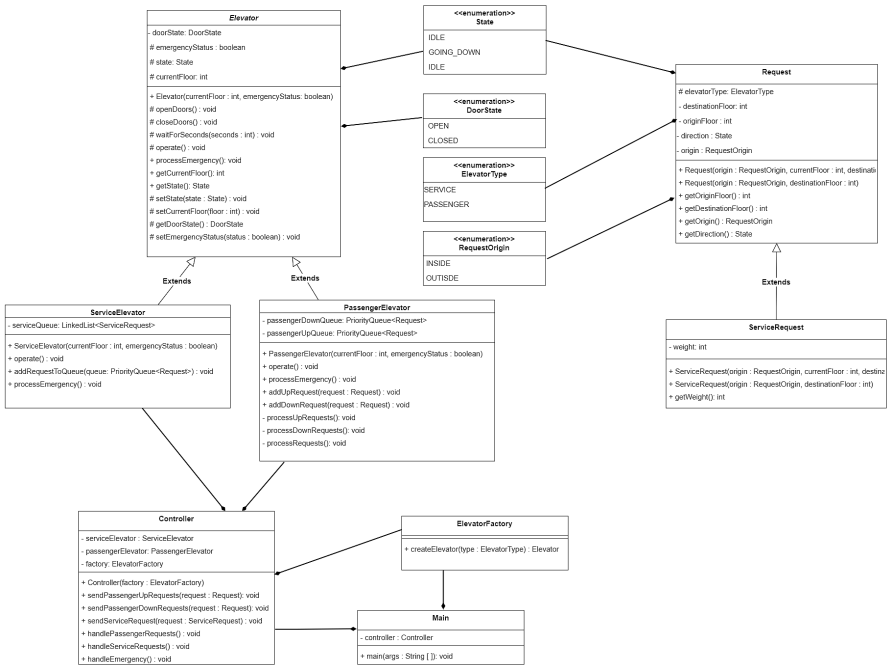
1. **Classes/Enums**: Represented by rectangles, and they're named to show the objects to be described in the system.
2. **Attributes**: Represented inside the class rectangle, below the class name.They represent the properties of the class.
3. **Operations/Methods**: Also represented inside the class rectangle, below the attributes. They

define what a class can do.

4. **Associations**: Represented by lines connecting classes. They indicate relationships between classes.

5. **Generalization (Inheritance)**: Represented by a line with a hollow arrowhead. It points from the subclass to the superclass, indicating an "is-a" relationship.

Elevator Design UML Diagram



Knowing all the above, we can now jump into the implementation.

Implementation

Using the above class diagram as a blueprint, we can proceed with our code implementation.

Imports:

Python **JavaScript** **C++**

```
from collections import deque
import heapq
import time
from enum import Enum
```

Our elevator can have three states: `GOING_UP, GOING_DOWN, IDLE`. For this, we can use an `enum` called `State`.

Python JavaScript Java C++

```
class State(Enum):
    IDLE = 1
    UP = 2
    DOWN = 3
    EMERGENCY = 4
```

Our building has two types of elevators: A passenger elevator and a service elevator, again, we can represent this using an enum called `ElevatorType`.

Python JavaScript Java C++

```
class ElevatorType(Enum):
    PASSENGER = 1
    SERVICE = 2
```

It is important to distinguish whether the request is made from the inside and outside the elevator. An enum can be used here called `RequestOrigin`

Python JavaScript Java C++

```
class RequestOrigin(Enum):
    INSIDE = 1
    OUTSIDE = 2
```

The elevator doors can also have two states: `OPEN, CLOSED`.

Python JavaScript C++

```
class DoorState(Enum):
    OPEN = 1
    CLOSED = 2
```

To call an elevator, the users need to press a button, either from inside or outside of the elevator. This sends out a request. For this, we can create a class called `Request`.

Python JavaScript Java C++

```
class Request:

    def __init__(self, origin,
origin_floor, destination_floor=None):
        self.origin = origin
        self.direction = State.IDLE
        self.origin_floor = origin_floor
        self.destination_floor =
destination_floor
        self.elevator_type =
ElevatorType.PASSENGER
```

```
        # Determine direction if both
origin_floor and destination_floor are
provided
        if destination_floor is not None:
            if origin_floor >
destination_floor:
                self.direction = State.DOWN
            elif origin_floor <
destination_floor:
                self.direction = State.UP

    def get_origin_floor(self):
        return self.origin_floor

    def get_destination_floor(self):
        return self.destination_floor

    def get_origin(self):
        return self.origin

    def get_direction(self):
        return self.direction

    # To determine order within the heap
    def __lt__(self, other):
        return self.destination_floor <
other.destination_floor
```

The service elevator is different to the passenger elevator in that it is made to move freight, furniture, heavy items etc. We can extend the `Request` class and create a `ServiceRequest` class.

Python JavaScript Java C++

```
class ServiceRequest(Request):

    def __init__(self, origin,
current_floor=None,
destination_floor=None):
        if current_floor is not None and
destination_floor is not None:
            super().__init__(origin,
current_floor, destination_floor)
        else:
            super().__init__(origin,
destination_floor)
        self.elevator_type =
ElevatorType.SERVICE
```

Requests will be handled by the elevators. We have two types of elevators, both of which share some attributes and methods, but also have additional functionality. Therefore, we can have an abstract class called `Elevator` to start with.

Python JavaScript Java C++

```
class Elevator:
    def __init__(self, current_floor,
emergency_status):
        self.current_floor = current_floor
        self.state = State.IDLE
        self.emergency_status =
emergency_status
        self.door_state = DoorState.CLOSED

    def open_doors(self):
        self.door_state = DoorState.OPEN
        print(f"Doors are OPEN on floor
{self.current_floor}")
```

```
    def close_doors(self):
        self.door_state = DoorState.CLOSED
        print("Doors are CLOSED")

    def wait_for_seconds(self, seconds):
        time.sleep(seconds)

    def operate(self):
        pass

    def process_emergency(self):
        pass

    def get_current_floor(self):
        return self.current_floor

    def get_state(self):
        return self.state

    def set_state(self, state):
        self.state = state

    def set_current_floor(self, floor):
        self.current_floor = floor

    def get_door_state(self):
        return self.door_state

    def set_emergency_status(self, status):
        self.emergency_status = status
```

We can now create our concrete elevators - `PassengerElevator` and `ServiceElevator` We can create a child class called `PassengerElevator`. This class uses a max and a min heap to prioritize direction of the requests.

Python JavaScript Java C++

```python
class PassengerElevator(Elevator):

    def __init__(self, current_floor,
emergency_status):
        super().__init__(current_floor,
emergency_status)
        self.passenger_up_queue = []
        self.passenger_down_queue = []

    def operate(self):
        while self.passenger_up_queue or
self.passenger_down_queue:
            self.process_requests()
        self.set_state(State.IDLE)
        print("All requests have been
fulfilled, elevator is now",
self.get_state())

    def process_emergency(self):
        self.passenger_up_queue.clear()
        self.passenger_down_queue.clear()
        self.set_current_floor(1)
        self.set_state(State.IDLE)
        self.open_doors()
        self.set_emergency_status(True)
        print("Queues cleared, current
floor is",
              self.get_current_floor(), ".
Doors are", self.get_door_state())

    def add_up_request(self, request):
        if request.get_origin() ==
RequestOrigin.OUTSIDE:
            pick_up_request =
Request(request.get_origin(
            ), request.get_origin_floor(),
request.get_origin_floor())
```

```
heapq.heappush(self.passenger_up_queue,
pick_up_request)

heapq.heappush(self.passenger_up_queue,
request)

    def add_down_request(self, request):
        if request.get_origin() ==
RequestOrigin.OUTSIDE:
            pick_up_request =
Request(request.get_origin(
            ), request.get_origin_floor(),
request.get_origin_floor())

heapq.heappush(self.passenger_down_queue,
pick_up_request)

heapq.heappush(self.passenger_down_queue,
request)

    def process_up_requests(self):
        while self.passenger_up_queue:
            up_request =
heapq.heappop(self.passenger_up_queue)

            if self.get_current_floor() ==
up_request.get_destination_floor():
                print("Currently on floor",
self.get_current_floor(),
                      ". No movement as
destination is the same.")
                continue
            print("The current floor is",
self.get_current_floor(),
                  ". Next stop:",
up_request.get_destination_floor())

            try:
                print("Moving ", end="")
                for _ in range(3):
```

```
                    print(".", end="",
flush=True)
                    time.sleep(0.5)  #
Pause for half a second between dots.
                time.sleep(1)  # Assuming 1
second to move to the next floor.
                print()
            except KeyboardInterrupt:
                pass
            except Exception as e:
                print("Error:", e)


self.set_current_floor(up_request.get_desti
nation_floor())
            print("Arrived at",
self.get_current_floor())

            self.open_doors()
            # Simulating 3 seconds for
people to enter/exit.
            self.wait_for_seconds(3)
            self.close_doors()

        print("Finished processing all the
up requests.")

    def process_down_requests(self):
        while self.passenger_down_queue:
            down_request =
heapq.heappop(self.passenger_down_queue)

            if self.get_current_floor() ==
down_request.get_destination_floor():
                print("Currently on floor",
self.get_current_floor(),
                      ". No movement as
destination is the same.")
                continue
```

```
            print("The current floor is",
self.get_current_floor(),
                  ". Next stop:",
down_request.get_destination_floor())

            try:
                print("Moving ", end="")
                for _ in range(3):
                    print(".", end="",
flush=True)
                    time.sleep(0.5)  #
Pause for half a second between dots.
                time.sleep(1)  # Assuming 1
second to move to the next floor.
                print()
            except KeyboardInterrupt:
                pass
            except Exception as e:
                print("Error:", e)


self.set_current_floor(down_request.get_des
tination_floor())
            print("Arrived at",
self.get_current_floor())

            self.open_doors()
            # Simulating 3 seconds for
people to enter/exit.
            self.wait_for_seconds(3)
            self.close_doors()

        print("Finished processing all the
down requests.")

    def process_requests(self):
        if self.get_state() == State.UP or
self.get_state() == State.IDLE:
            self.process_up_requests()
            if self.passenger_down_queue:
```

```
                print("Now processing down
requests...")

self.process_down_requests()
        else:
            self.process_down_requests()
            if self.passenger_up_queue:
                print("Now processing up
requests...")
                self.process_up_requests()
```

The second concrete class is `ServiceElevator`, which operates on a first-come first-serve basis.

Python JavaScript Java C++

```
class ServiceElevator(Elevator):

    def __init__(self, current_floor,
emergency_status):
        super().__init__(current_floor,
emergency_status)
        self.service_queue = deque()

    def operate(self):
        while self.service_queue:
            curr_request =
self.service_queue.popleft()

            print()  # Move to the next
line after the dots.
            print("Currently at",
self.get_current_floor())
            try:
                time.sleep(1)  # Assuming 1
second to move to the next floor.

print(curr_request.get_direction(), end="")
```

```
                for _ in range(3):
                    print(".", end="",
flush=True)
                    time.sleep(0.5)  #
Pause for half a second between dots.
            except KeyboardInterrupt:
                pass
            except Exception as e:
                print("Error:", e)


self.set_current_floor(curr_request.get_des
tination_floor())

self.set_state(curr_request.get_direction()
)
            print("Arrived at",
self.get_current_floor())

            self.open_doors()
            # Simulating 3 seconds for
loading/unloading.
            self.wait_for_seconds(3)
            self.close_doors()

        self.set_state(State.IDLE)
        print("All requests have been
fulfilled, elevator is now",
self.get_state())

    def add_request_to_queue(self,
request):
        self.service_queue.append(request)

    def process_emergency(self):
        self.service_queue.clear()
        self.set_current_floor(1)
        self.set_state(State.IDLE)
        self.open_doors()
        self.set_emergency_status(True)
```

```
            print("Queue cleared, current floor
is", self.get_current_floor(),
                  ". Doors are",
self.get_door_state())
```

The user does not need to know all of the above business logic. To abstract the instantiation, we can use the **Factory** pattern in our `ElevatorFactory` class.

Python JavaScript Java C++

```
class ElevatorFactory:
    @staticmethod
    def create_elevator(elevator_type:
ElevatorType):
        if elevator_type ==
ElevatorType.PASSENGER:
            return PassengerElevator(1,
False)
        elif elevator_type ==
ElevatorType.SERVICE:
            return ServiceElevator(1,
False)
        else:
            return None
```

Now that we have all of the above set up, we can go ahead and and add a `Controller` class. The `Controller` class is the class that the user interacts with.

Python JavaScript Java C++

```
class Controller:

    def __init__(self, factory):
        self.factory = factory
        self.passenger_elevator =
factory.create_elevator(
            ElevatorType.PASSENGER)
        self.service_elevator =
factory.create_elevator(ElevatorType.SERVIC
E)

    def send_passenger_up_requests(self,
request):

self.passenger_elevator.add_up_request(requ
est)

    def send_passenger_down_requests(self,
request):

self.passenger_elevator.add_down_request(re
quest)

    def send_service_request(self,
request):

self.service_elevator.add_request_to_queue(
request)

    def handle_passenger_requests(self):
        self.passenger_elevator.operate()

    def handle_service_requests(self):
        self.service_elevator.operate()

    def handle_emergency(self):

self.passenger_elevator.process_emergency()
```

```
self.service_elevator.process_emergency()
```

Finally, we can have a `Main` class which contains our main method, instantiates the `Controller` and sends the requests to the pertinent elevators.

Python JavaScript Java C++

```
class Main:

    @staticmethod
    def main():
        factory = ElevatorFactory()
        controller = Controller(factory)


controller.send_passenger_up_requests(
            Request(RequestOrigin.OUTSIDE,
1, 5)
        )

controller.send_passenger_down_requests(
            Request(RequestOrigin.OUTSIDE,
4, 2)
        )

controller.send_passenger_up_requests(
            Request(RequestOrigin.OUTSIDE,
3, 6)
        )

controller.handle_passenger_requests()


controller.send_passenger_up_requests(
            Request(RequestOrigin.OUTSIDE,
1, 9)
```

```
        )

controller.send_passenger_down_requests(
            Request(RequestOrigin.INSIDE,
5))

controller.send_passenger_up_requests(
            Request(RequestOrigin.OUTSIDE,
4, 12)
        )

controller.send_passenger_down_requests(
            Request(RequestOrigin.OUTSIDE,
10, 2)
        )

controller.handle_passenger_requests()

        print("Now processing service
requests")

        controller.send_service_request(

ServiceRequest(RequestOrigin.INSIDE, 13))
        controller.send_service_request(

ServiceRequest(RequestOrigin.OUTSIDE, 13,
2)
        )
        controller.send_service_request(

ServiceRequest(RequestOrigin.INSIDE, 13,
15)
        )


controller.handle_service_requests()
```

```
if __name__ == "__main__":
    Main.main()
```

Object-Oriented Design Interview

0 / 7

Interview Questions

6 - Design Chess

10:34

Mark Lesson Complete View Code

Design Chess

Background

Chess is played on an 8x8 board and involves two players: one controlling the Black pieces and the other controlling the White pieces. Each player starts with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The primary goal of the game is to checkmate the opponent's king, placing the king in a position where it cannot escape capture. Each of the six different types of pieces has its own unique way of moving.

Requirements

Board Layout and Basics

- How is the 8x8 chessboard represented in the system?

- How are the 64 squares on the board identified? (For instance, are they labeled as A1, A2, ... H8?)
- How are the pieces initialized on the board at the beginning of a game?

Piece Movement and Capturing

- How does each type of chess piece move (Pawn, Rook, Knight, Bishop, Queen, King)?
- How does the system verify the validity of a move for a given piece?
- How are special moves like 'en passant', 'castling', and 'pawn promotion' handled?

Game Progression and Turn Mechanics

- Do the players take turns?
- How are illegal moves, like moving into check, prevented?

User Interaction and Experience

- How do players indicate the piece they want to move and where they want to move it to?
- How is the game visually represented to the players?
- What feedback is given to users if an invalid move is made?

Functional Requirements

Board Layout and Basics

- The board can be represented using a 2-D array.
- Black pawns are placed on the second row, white pawns on the seventh row. The other black pieces occupy the first row, and the other white pieces occupy the eighth row.

Piece Movement and Capturing

- In the game, each piece moves as follows:

    1. Pawns move forward one square. They may move two squares if and only if it is their first move.

    2. Rooks move in straight lines, either horizontally or vertically.

    3. Knights move in an 'L'-shape: two squares in one direction (horizontally or vertically) followed by one square perpendicular to that, or vice-versa. The knight is the only piece that can jump over other pieces.

    4. Bishops move diagonally on the board. They also must stay on the same color square all along. The bishop cannot jump over pieces.

    5. Queens move in straight lines in any direction: horizontally, vertically, or diagonally. The queen cannot jump over pieces.

    6. Kings move one square in any direction. The king cannot jump over pieces.

Game Progression and Turn Mechanics

- The game should ensure that players take turns.

User Interaction and Experience

- The game should prompt the user to enter both the starting and destination rows and columns.

Non-Functional Requirements

Board Layout and Basics

- For simplicity and ease of implementation, each square can be accessed by using the respective row and

column.

Piece Movement and Capturing

- To keep the design simple, special moves like 'en passant', 'castling', and 'pawn promotion' do not need to be implemented.
- The implementation should prevent the following invalid moves:
    - Moving into check.
    - Allowing players to move out of turn.
    - Allowing players to capture their own pieces.
    - Letting players move beyond the boundaries of the 8x8 board.

Game Progression and Turn Mechanics

- The game should maintain the state of the board and include validity checks to prevent illegal moves.

User Interaction and Experience

- A terminal application UI is required.
- The game should inform the user if they attempt to move an enemy piece, if their input is outside the 8x8 grid, or if the move is invalid for the selected piece.

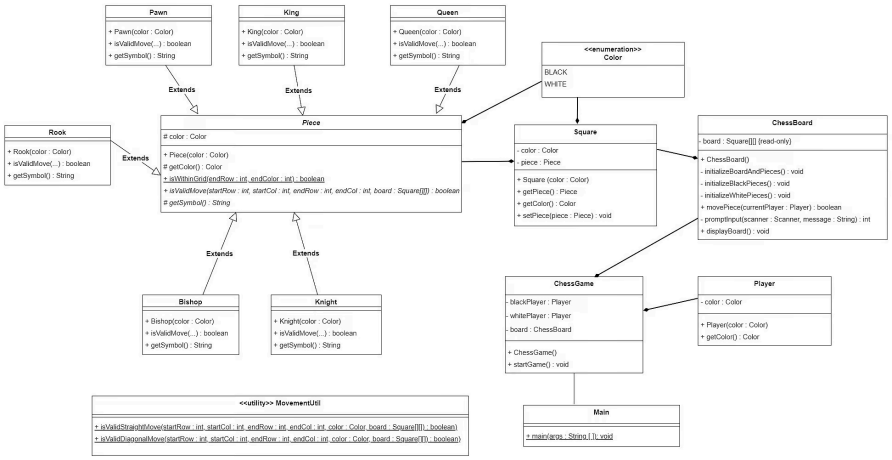
Design and Implementation

> **NOTE:** If we were to create a fully functional chess implementation, it would require much more code than what we present below. For the sake of brevity, game-winning conditions like stalemate and checkmate have been left out.

Design

To recall, the UML class diagram below represents the overall structure of our implementation. This design targets to strike a balance between a fully functional chess game with validating moves, while omitting any special moves like en passant or king's castling.

Chess Design UML Diagram



Code

Using the class diagram provided above as a blueprint, we can proceed to develop our code for the chess game implementation.

Since the chess board only features pieces in black and white, we can use an enum named Color to represent them.

Python JavaScript Java C++

```
class Color(Enum):
    WHITE = 1
    BLACK = 2
```

A chessboard is made up of squares, each of which has a color and may contain a piece. To facilitate this, we can create a Square class. This class will be useful when we set up the chessboard.

Python JavaScript Java C++

```
class Square:
    def __init__(self, color):
        self.color = color
        self.piece = None

    def get_piece(self):
        return self.piece

    def get_color(self):
        return self.color

    def set_piece(self, piece):
        self.piece = piece
```

The next natural step is to create all six types of chess pieces. Since each piece has a color, a unique symbol, and its own set of movement rules, we can create an abstract class called `Piece`. This class will be extended by our Rook, Bishop, Knight, King, Queen, and Pawn classes.

Python JavaScript Java C++

```
class Piece(ABC):
    def __init__(self, color):
        self.color = color
```

```
    def get_color(self):
        return self.color

    @staticmethod
    def is_within_grid(end_row, end_col):
        return 0 <= end_row <= 7 and 0 <=
end_col <= 7

    @abstractmethod
    def is_valid_move(self, start_row,
start_col, end_row, end_col, board):
        pass

    @abstractmethod
    def get_symbol(self):
        pass
```

And now the concrete implementations of Piece:

Pawn

Python JavaScript Java C++

```
class Pawn(Piece):
    def is_valid_move(self, start_row,
start_col, end_row, end_col, board):
        if not
Piece.is_within_grid(end_row, end_col):
            return False
        row_movement = end_row - start_row
        col_movement = end_col - start_col
        direction = -1 if self.get_color()
== Color.WHITE else 1

        if col_movement != 0:
            return False

        if row_movement == direction and
```

```
board[end_row][end_col].get_piece() is
None:
            return True

        if (self.is_first_move(start_row)
and row_movement == (2 * direction) and
            board[start_row + direction]
[start_col].get_piece() is None and
            board[end_row]
[end_col].get_piece() is None):
            return True

        return False

    def get_symbol(self):
        return "P" if self.get_color() ==
Color.WHITE else "p"

    def is_first_move(self, start_row):
        return (self.get_color() ==
Color.WHITE and start_row == 6) or \
               (self.get_color() ==
Color.BLACK and start_row == 1)
```

Knight

Python JavaScript Java C++

```
class Knight(Piece):
    def is_valid_move(self, start_row,
start_col, end_row, end_col, board):
        if not
Piece.is_within_grid(end_row, end_col):
            return False
        row_movement = abs(end_row -
start_row)
        col_movement = abs(end_col -
start_col)
```

```
        if row_movement == 2 and
col_movement == 1 or \
           row_movement == 1 and
col_movement == 2:

            if board[end_row]
[end_col].get_piece() is not None and \
               board[end_row]
[end_col].get_piece().get_color() ==
self.get_color():
                return False

            return True

        return False

    def get_symbol(self):
        return "N" if self.get_color() ==
Color.WHITE else "n"
```

> Since queen combines the power of the rook and bishop, we can create a utility class called `MovementUtil` to make our design more robust.

MovementUtil

Python JavaScript Java C++

```
class MovementUtil:

    @staticmethod
    def is_valid_straight_move(start_row,
start_col, end_row, end_col, color, board):
        if not
Piece.is_within_grid(end_row, end_col):
            return False
```

```
        row_movement = abs(end_row -
start_row)
        col_movement = abs(end_col -
start_col)

        if row_movement != 0 and
col_movement != 0 or (row_movement == 0 and
col_movement == 0):
            return False
        else:
            row_increment = 1 if end_row >
start_row else -1
            col_increment = 1 if end_col >
start_col else -1

            if row_movement == 0:
                y = start_col +
col_increment
                while y != end_col:
                    if board[start_row]
[y].get_piece() is not None:
                        return False
                    y += col_increment
            else:
                x = start_row +
row_increment
                while x != end_row:
                    if board[x]
[start_col].get_piece() is not None:
                        return False
                    x += row_increment

            if board[end_row]
[end_col].get_piece() is not None and \
               board[end_row]
[end_col].get_piece().get_color() == color:
                return False

            return True
```

```
    @staticmethod
    def is_valid_diagonal_move(start_row,
start_col, end_row, end_col, color, board):
        if not
Piece.is_within_grid(end_row, end_col):
            return False

        row_movement = abs(end_row -
start_row)
        col_movement = abs(end_col -
start_col)

        if row_movement == 0 or
col_movement == 0:
            return False

        if row_movement == col_movement:
            row_increment = 1 if end_row >
start_row else -1
            col_increment = 1 if end_col >
start_col else -1

            x, y = start_row +
row_increment, start_col + col_increment

            while x != end_row and y !=
end_col:
                if board[x][y].get_piece()
is not None:
                    return False

                x += row_increment
                y += col_increment

            if board[end_row]
[end_col].get_piece() is not None and \
                board[end_row]
[end_col].get_piece().get_color() == color:
                return False
```

```
            return True
        else:
            return False
```

Rook

Python JavaScript Java C++

```
class Rook(Piece):
    def is_valid_move(self, start_row,
start_col, end_row, end_col, board):
        return
MovementUtil.is_valid_straight_move(start_r
ow, start_col, end_row, end_col,
self.get_color(), board)

    def get_symbol(self):
        return 'R' if self.get_color() ==
Color.WHITE else 'r'
```

Bishop

Python JavaScript Java C++

```
class Bishop(Piece):
    def is_valid_move(self, start_row,
start_col, end_row, end_col, board):
        return
MovementUtil.is_valid_diagonal_move(start_r
ow, start_col, end_row, end_col,
self.get_color(), board)

    def get_symbol(self):
```

```
        return 'B' if self.get_color() ==
Color.WHITE else 'b'
```

King

Python JavaScript Java C++

```
class King(Piece):
    def is_valid_move(self, start_row,
start_col, end_row, end_col, board):
        if not
Piece.is_within_grid(end_row, end_col):
            return False

        row_movement = abs(end_row -
start_row)
        col_movement = abs(end_col -
start_col)

        if row_movement > 1 or col_movement
> 1:
            return False

        if board[end_row]
[end_col].get_piece() is not None and \
           board[end_row]
[end_col].get_piece().get_color() ==
self.get_color():
            return False

        return True

    def get_symbol(self):
        return 'K' if self.get_color() ==
Color.WHITE else 'k'
```

Queen

Python JavaScript Java C++

```
class Queen(Piece):
    def is_valid_move(self, start_row,
start_col, end_row, end_col, board):
        return
MovementUtil.is_valid_straight_move(start_r
ow, start_col, end_row, end_col,
self.get_color(), board) or \

MovementUtil.is_valid_diagonal_move(start_r
ow, start_col, end_row, end_col,
self.get_color(), board)

    def get_symbol(self):
        return 'Q' if self.get_color() ==
Color.WHITE else 'q'
```

Now we have all the required classes to create our ChessBoard class. The `ChessBoard` will initialize the 64 squares and place all the pieces in their initial positions. It is also responsible for managing the state of the board.

Python JavaScript Java C++

```
class ChessBoard:

    def __init__(self):
        self.board = [[None for _ in
range(8)] for _ in range(8)]
        self.initialize_board_and_pieces()

    def initialize_board_and_pieces(self):
        for i in range(8):
            for j in range(8):
```

```
                square_color = Color.BLACK
if (i + j) % 2 == 0 else Color.WHITE
                self.board[i][j] =
Square(square_color)

        self.initialize_black_pieces()
        self.initialize_white_pieces()

    def initialize_black_pieces(self):
        for i in range(8):
            self.board[1]
[i].set_piece(Pawn(Color.BLACK))

        self.board[0]
[0].set_piece(Rook(Color.BLACK))
        self.board[0]
[7].set_piece(Rook(Color.BLACK))
        self.board[0]
[1].set_piece(Knight(Color.BLACK))
        self.board[0]
[6].set_piece(Knight(Color.BLACK))
        self.board[0]
[2].set_piece(Bishop(Color.BLACK))
        self.board[0]
[5].set_piece(Bishop(Color.BLACK))
        self.board[0]
[3].set_piece(Queen(Color.BLACK))
        self.board[0]
[4].set_piece(King(Color.BLACK))

    def initialize_white_pieces(self):
        for i in range(8):
            self.board[6]
[i].set_piece(Pawn(Color.WHITE))

        self.board[7]
[0].set_piece(Rook(Color.WHITE))
        self.board[7]
[7].set_piece(Rook(Color.WHITE))
        self.board[7]
```

```
[1].set_piece(Knight(Color.WHITE))
        self.board[7]
[6].set_piece(Knight(Color.WHITE))
        self.board[7]
[2].set_piece(Bishop(Color.WHITE))
        self.board[7]
[5].set_piece(Bishop(Color.WHITE))
        self.board[7]
[3].set_piece(Queen(Color.WHITE))
        self.board[7]
[4].set_piece(King(Color.WHITE))

    def move_piece(self, current_player):
        while True:
            start_row = int(input("Enter
starting row: "))
            start_col = int(input("Enter
starting column: "))
            end_row = int(input("Enter
destination row: "))
            end_col = int(input("Enter
destination column: "))

            if not
Piece.is_within_grid(end_row, end_col):
                return False

            piece_to_move =
self.board[start_row]
[start_col].get_piece()

            if not piece_to_move:
                print("There's no piece at
the specified starting position.")
                continue

            if piece_to_move.color !=
current_player.color:
                print("It's not your turn
to move this piece.")
```

```
                continue

            if
piece_to_move.is_valid_move(start_row,
start_col, end_row, end_col, self.board):
                destination_piece =
self.board[end_row][end_col].get_piece()
                if destination_piece and
destination_piece.color !=
piece_to_move.color:
                    self.board[end_row]
[end_col].set_piece(None)

                self.board[end_row]
[end_col].set_piece(piece_to_move)
                self.board[start_row]
[start_col].set_piece(None)  # Clear the
original square
                print(f"
{piece_to_move.get_symbol()} moved to
{end_row}, {end_col}")
                return True
            else:
                print(f"Invalid move for
the {piece_to_move.get_symbol()}. Please
try again.")

    def display_board(self):
        print("  0 1 2 3 4 5 6 7")
        print("  ---------------")
        for i in range(8):
            print(i, end='|')
            for j in range(8):
                piece = self.board[i]
[j].get_piece()
                print(piece.get_symbol() +
" " if piece else ". ", end="")
            print()
```

Now that we have a chessboard set up with all the pieces in their initial positions, our next step is to create a `Player` class.

Python JavaScript Java C++

```
class Player:
    def __init__(self, color):
        self.color = color

    def get_color(self):
        return self.color
```

To bring it altogether, our `ChessGame` class will be responsible for bringing the `ChessBoard` and the `Player` class together and will handle initiation of a new game. Again, for brevity, we have decided to omit game-ending conditions and from a design perspective, it is not that important.

Python JavaScript Java C++

```
class ChessGame:
    def __init__(self):
        self.board = ChessBoard()
        self.white_player =
Player(Color.WHITE)
        self.black_player =
Player(Color.BLACK)
        self.current_player =
self.white_player

    def start_game(self):
        print("Welcome to Chess, UPPERCASE
```

```
denotes white pieces, LOWERCASE denotes
black pieces.")
        self.board.display_board()

        current_player = self.white_player

        while True:
            print("Current turn:" +
str(current_player.get_color()))

            move_successful =
self.board.move_piece(self.current_player)
            if move_successful:
                self.board.display_board()
                self.current_player =
self.black_player if self.current_player ==
self.white_player else self.white_player
            else:
                print("Invalid move. Please
try again.")
```

Finally, we can instantiate a new game:

Python JavaScript Java C++

```
game = ChessGame()
game.start_game()
```

The below output will be shown in the terminal upon successful compilation of the program.

```
Welcome to Chess, UPPERCASE denotes white,
LOWERCASE denotes black
0 1 2 3 4 5 6 7

---
```

```
0|r n b q k b n r
1|p p p p p p p p
2|. . . . . . . .
3|. . . . . . . .
4|. . . . . . . .
5|. . . . . . . .
6|P P P P P P P P
7|R N B Q K B N R
Current turn: White
Enter starting row:
```