**Design a call center**

**Constraints and assumptions**

* What levels of employees are in the call center?
  + Operator, supervisor, director
* Can we assume operators always get the initial calls?
  + Yes
* If there is no free operators or the operator can't handle the call, does the call go to the supervisors?
  + Yes
* If there is no free supervisors or the supervisor can't handle the call, does the call go to the directors?
  + Yes
* Can we assume the directors can handle all calls?
  + Yes
* What happens if nobody can answer the call?
  + It gets queued
* Do we need to handle 'VIP' calls where we put someone to the front of the line?
  + No
* Can we assume inputs are valid or do we have to validate them?
  + Assume they're valid

**Solution**

In [1]:

%%writefile call\_center.py

from abc import ABCMeta, abstractmethod

from collections import deque

from enum import Enum

class Rank(Enum):

OPERATOR = 0

SUPERVISOR = 1

DIRECTOR = 2

class Employee(metaclass=ABCMeta):

def \_\_init\_\_(self, employee\_id, name, rank, call\_center):

self.employee\_id = employee\_id

self.name = name

self.rank = rank

self.call = None

self.call\_center = call\_center

def take\_call(self, call):

"""Assume the employee will always successfully take the call."""

self.call = call

self.call.employee = self

self.call.state = CallState.IN\_PROGRESS

def complete\_call(self):

self.call.state = CallState.COMPLETE

self.call\_center.notify\_call\_completed(self.call)

@abstractmethod

def escalate\_call(self):

pass

def \_escalate\_call(self):

self.call.state = CallState.READY

call = self.call

self.call = None

self.call\_center.notify\_call\_escalated(call)

class Operator(Employee):

def \_\_init\_\_(self, employee\_id, name):

super(Operator, self).\_\_init\_\_(employee\_id, name, Rank.OPERATOR)

def escalate\_call(self):

self.call.level = Rank.SUPERVISOR

self.\_escalate\_call()

class Supervisor(Employee):

def \_\_init\_\_(self, employee\_id, name):

super(Operator, self).\_\_init\_\_(employee\_id, name, Rank.SUPERVISOR)

def escalate\_call(self):

self.call.level = Rank.DIRECTOR

self.\_escalate\_call()

class Director(Employee):

def \_\_init\_\_(self, employee\_id, name):

super(Operator, self).\_\_init\_\_(employee\_id, name, Rank.DIRECTOR)

def escalate\_call(self):

raise NotImplemented('Directors must be able to handle any call')

class CallState(Enum):

READY = 0

IN\_PROGRESS = 1

COMPLETE = 2

class Call(object):

def \_\_init\_\_(self, rank):

self.state = CallState.READY

self.rank = rank

self.employee = None

class CallCenter(object):

def \_\_init\_\_(self, operators, supervisors, directors):

self.operators = operators

self.supervisors = supervisors

self.directors = directors

self.queued\_calls = deque()

def dispatch\_call(self, call):

if call.rank not in (Rank.OPERATOR, Rank.SUPERVISOR, Rank.DIRECTOR):

raise ValueError('Invalid call rank: {}'.format(call.rank))

employee = None

if call.rank == Rank.OPERATOR:

employee = self.\_dispatch\_call(call, self.operators)

if call.rank == Rank.SUPERVISOR or employee is None:

employee = self.\_dispatch\_call(call, self.supervisors)

if call.rank == Rank.DIRECTOR or employee is None:

employee = self.\_dispatch\_call(call, self.directors)

if employee is None:

self.queued\_calls.append(call)

def \_dispatch\_call(self, call, employees):

for employee in employees:

if employee.call is None:

employee.take\_call(call)

return employee

return None

def notify\_call\_escalated(self, call): # ...

def notify\_call\_completed(self, call): # ...

def dispatch\_queued\_call\_to\_newly\_freed\_employee(self, call, employee): # ...

Overwriting call\_center.py

**Design a deck of cards**

**Constraints and assumptions**

* Is this a generic deck of cards for games like poker and black jack?
  + Yes, design a generic deck then extend it to black jack
* Can we assume the deck has 52 cards (2-10, Jack, Queen, King, Ace) and 4 suits?
  + Yes
* Can we assume inputs are valid or do we have to validate them?
  + Assume they're valid

**Solution**

In [1]:

%%writefile deck\_of\_cards.py

from abc import ABCMeta, abstractmethod

from enum import Enum

import sys

class Suit(Enum):

HEART = 0

DIAMOND = 1

CLUBS = 2

SPADE = 3

class Card(metaclass=ABCMeta):

def \_\_init\_\_(self, value, suit):

self.value = value

self.suit = suit

self.is\_available = True

@property

@abstractmethod

def value(self):

pass

@value.setter

@abstractmethod

def value(self, other):

pass

class BlackJackCard(Card):

def \_\_init\_\_(self, value, suit):

super(BlackJackCard, self).\_\_init\_\_(value, suit)

def is\_ace(self):

return True if self.\_value == 1 else False

def is\_face\_card(self):

"""Jack = 11, Queen = 12, King = 13"""

return True if 10 < self.\_value <= 13 else False

@property

def value(self):

if self.is\_ace() == 1:

return 1

elif self.is\_face\_card():

return 10

else:

return self.\_value

@value.setter

def value(self, new\_value):

if 1 <= new\_value <= 13:

self.\_value = new\_value

else:

raise ValueError('Invalid card value: {}'.format(new\_value))

class Hand(object):

def \_\_init\_\_(self, cards):

self.cards = cards

def add\_card(self, card):

self.cards.append(card)

def score(self):

total\_value = 0

for card in card:

total\_value += card.value

return total\_value

class BlackJackHand(Hand):

BLACKJACK = 21

def \_\_init\_\_(self, cards):

super(BlackJackHand, self).\_\_init\_\_(cards)

def score(self):

min\_over = sys.MAXSIZE

max\_under = -sys.MAXSIZE

for score in self.possible\_scores():

if self.BLACKJACK < score < min\_over:

min\_over = score

elif max\_under < score <= self.BLACKJACK:

max\_under = score

return max\_under if max\_under != -sys.MAXSIZE else min\_over

def possible\_scores(self):

"""Return a list of possible scores, taking Aces into account."""

# ...

class Deck(object):

def \_\_init\_\_(self, cards):

self.cards = cards

self.deal\_index = 0

def remaining\_cards(self):

return len(self.cards) - deal\_index

def deal\_card():

try:

card = self.cards[self.deal\_index]

card.is\_available = False

self.deal\_index += 1

except IndexError:

return None

return card

def shuffle(self): # ...

Overwriting deck\_of\_cards.py

**Design a hash map**

**Constraints and assumptions**

* For simplicity, are the keys integers only?
  + Yes
* For collision resolution, can we use chaining?
  + Yes
* Do we have to worry about load factors?
  + No
* Can we assume inputs are valid or do we have to validate them?
  + Assume they're valid
* Can we assume this fits memory?
  + Yes

**Solution**

In [1]:

%%writefile hash\_map.py

class Item(object):

def \_\_init\_\_(self, key, value):

self.key = key

self.value = value

class HashTable(object):

def \_\_init\_\_(self, size):

self.size = size

self.table = [[] for \_ in range(self.size)]

def \_hash\_function(self, key):

return key % self.size

def set(self, key, value):

hash\_index = self.\_hash\_function(key)

for item in self.table[hash\_index]:

if item.key == key:

item.value = value

return

self.table[hash\_index].append(Item(key, value))

def get(self, key):

hash\_index = self.\_hash\_function(key)

for item in self.table[hash\_index]:

if item.key == key:

return item.value

raise KeyError('Key not found')

def remove(self, key):

hash\_index = self.\_hash\_function(key)

for index, item in enumerate(self.table[hash\_index]):

if item.key == key:

del self.table[hash\_index][index]

return

raise KeyError('Key not found')

Overwriting hash\_map.py

**Design an LRU cache**

**Constraints and assumptions**

* What are we caching?
  + We are cahing the results of web queries
* Can we assume inputs are valid or do we have to validate them?
  + Assume they're valid
* Can we assume this fits memory?
  + Yes

**Solution**

In [1]:

%%writefile lru\_cache.py

class Node(object):

def \_\_init\_\_(self, results):

self.results = results

self.prev = None

self.next = None

class LinkedList(object):

def \_\_init\_\_(self):

self.head = None

self.tail = None

def move\_to\_front(self, node): # ...

def append\_to\_front(self, node): # ...

def remove\_from\_tail(self): # ...

class Cache(object):

def \_\_init\_\_(self, MAX\_SIZE):

self.MAX\_SIZE = MAX\_SIZE

self.size = 0

self.lookup = {} # key: query, value: node

self.linked\_list = LinkedList()

def get(self, query)

"""Get the stored query result from the cache.

Accessing a node updates its position to the front of the LRU list.

"""

node = self.lookup[query]

if node is None:

return None

self.linked\_list.move\_to\_front(node)

return node.results

def set(self, results, query):

"""Set the result for the given query key in the cache.

When updating an entry, updates its position to the front of the LRU list.

If the entry is new and the cache is at capacity, removes the oldest entry

before the new entry is added.

"""

node = self.lookup[query]

if node is not None:

# Key exists in cache, update the value

node.results = results

self.linked\_list.move\_to\_front(node)

else:

# Key does not exist in cache

if self.size == self.MAX\_SIZE:

# Remove the oldest entry from the linked list and lookup

self.lookup.pop(self.linked\_list.tail.query, None)

self.linked\_list.remove\_from\_tail()

else:

self.size += 1

# Add the new key and value

new\_node = Node(results)

self.linked\_list.append\_to\_front(new\_node)

self.lookup[query] = new\_node

Overwriting lru\_cache.py

**Design an online chat**

**Constraints and assumptions**

* Assume we'll focus on the following workflows:
  + Text conversations only
  + Users
    - Add a user
    - Remove a user
    - Update a user
    - Add to a user's friends list
      * Add friend request
        + Approve friend request
        + Reject friend request
    - Remove from a user's friends list
  + Create a group chat
    - Invite friends to a group chat
    - Post a message to a group chat
  + Private 1-1 chat
    - Invite a friend to a private chat
    - Post a meesage to a private chat
* No need to worry about scaling initially

**Solution**

In [1]:

%%writefile online\_chat.py

from abc import ABCMeta

class UserService(object):

def \_\_init\_\_(self):

self.users\_by\_id = {} # key: user id, value: User

def add\_user(self, user\_id, name, pass\_hash): # ...

def remove\_user(self, user\_id): # ...

def add\_friend\_request(self, from\_user\_id, to\_user\_id): # ...

def approve\_friend\_request(self, from\_user\_id, to\_user\_id): # ...

def reject\_friend\_request(self, from\_user\_id, to\_user\_id): # ...

class User(object):

def \_\_init\_\_(self, user\_id, name, pass\_hash):

self.user\_id = user\_id

self.name = name

self.pass\_hash = pass\_hash

self.friends\_by\_id = {} # key: friend id, value: User

self.friend\_ids\_to\_private\_chats = {} # key: friend id, value: private chats

self.group\_chats\_by\_id = {} # key: chat id, value: GroupChat

self.received\_friend\_requests\_by\_friend\_id = {} # key: friend id, value: AddRequest

self.sent\_friend\_requests\_by\_friend\_id = {} # key: friend id, value: AddRequest

def message\_user(self, friend\_id, message): # ...

def message\_group(self, group\_id, message): # ...

def send\_friend\_request(self, friend\_id): # ...

def receive\_friend\_request(self, friend\_id): # ...

def approve\_friend\_request(self, friend\_id): # ...

def reject\_friend\_request(self, friend\_id): # ...

class Chat(metaclass=ABCMeta):

def \_\_init\_\_(self, chat\_id):

self.chat\_id = chat\_id

self.users = []

self.messages = []

class PrivateChat(Chat):

def \_\_init\_\_(self, first\_user, second\_user):

super(PrivateChat, self).\_\_init\_\_()

self.users.append(first\_user)

self.users.append(second\_user)

class GroupChat(Chat):

def add\_user(self, user): # ...

def remove\_user(self, user): # ...

class Message(object):

def \_\_init\_\_(self, message\_id, message, timestamp):

self.message\_id = message\_id

self.message = message

self.timestamp = timestamp

class AddRequest(object):

def \_\_init\_\_(self, from\_user\_id, to\_user\_id, request\_status, timestamp):

self.from\_user\_id = from\_user\_id

self.to\_user\_id = to\_user\_id

self.request\_status = request\_status

self.timestamp = timestamp

class RequestStatus(Enum):

UNREAD = 0

READ = 1

ACCEPTED = 2

REJECTED = 3

Overwriting online\_chat.py

**Design a parking lot**

**Constraints and assumptions**

* What types of vehicles should we support?
  + Motorcycle, Car, Bus
* Does each vehicle type take up a different amount of parking spots?
  + Yes
  + Motorcycle spot -> Motorcycle
  + Compact spot -> Motorcycle, Car
  + Large spot -> Motorcycle, Car
  + Bus can park if we have 5 consecutive "large" spots
* Does the parking lot have multiple levels?
  + Yes

**Solution**

In [1]:

%%writefile parking\_lot.py

from abc import ABCMeta, abstractmethod

class VehicleSize(Enum):

MOTORCYCLE = 0

COMPACT = 1

LARGE = 2

class Vehicle(metaclass=ABCMeta):

def \_\_init\_\_(self, vehicle\_size, license\_plate, spot\_size):

self.vehicle\_size = vehicle\_size

self.license\_plate = license\_plate

self.spot\_size = spot\_size

self.spots\_taken = []

def clear\_spots(self):

for spot in self.spots\_taken:

spot.remove\_vehicle(self)

self.spots\_taken = []

def take\_spot(self, spot):

self.spots\_taken.append(spot)

@abstractmethod

def can\_fit\_in\_spot(self, spot):

pass

class Motorcycle(Vehicle):

def \_\_init\_\_(self, license\_plate):

super(Motorcycle, self).\_\_init\_\_(VehicleSize.MOTORCYCLE, license\_plate, spot\_size=1)

def can\_fit\_in\_spot(self, spot):

return True

class Car(Vehicle):

def \_\_init\_\_(self, license\_plate):

super(Car, self).\_\_init\_\_(VehicleSize.COMPACT, license\_plate, spot\_size=1)

def can\_fit\_in\_spot(self, spot):

return True if (spot.size == LARGE or spot.size == COMPACT) else False

class Bus(Vehicle):

def \_\_init\_\_(self, license\_plate):

super(Bus, self).\_\_init\_\_(VehicleSize.LARGE, license\_plate, spot\_size=5)

def can\_fit\_in\_spot(self, spot):

return True if spot.size == LARGE else False

class ParkingLot(object):

def \_\_init\_\_(self, num\_levels):

self.num\_levels = num\_levels

self.levels = []

def park\_vehicle(self, vehicle):

for level in levels:

if level.park\_vehicle(vehicle):

return True

return False

class Level(object):

SPOTS\_PER\_ROW = 10

def \_\_init\_\_(self, floor, total\_spots):

self.floor = floor

self.num\_spots = total\_spots

self.available\_spots = 0

self.parking\_spots = []

def spot\_freed(self):

self.available\_spots += 1

def park\_vehicle(self, vehicle):

spot = self.\_find\_available\_spot(vehicle)

if spot is None:

return None

else:

spot.park\_vehicle(vehicle)

return spot

def \_find\_available\_spot(self, vehicle):

"""Find an available spot where vehicle can fit, or return None"""

# ...

def \_park\_starting\_at\_spot(self, spot, vehicle):

"""Occupy starting at spot.spot\_number to vehicle.spot\_size."""

# ...

class ParkingSpot(object):

def \_\_init\_\_(self, level, row, spot\_number, spot\_size, vehicle\_size):

self.level = level

self.row = row

self.spot\_number = spot\_number

self.spot\_size = spot\_size

self.vehicle\_size = vehicle\_size

self.vehicle = None

def is\_available(self):

return True if self.vehicle is None else False

def can\_fit\_vehicle(self, vehicle):

if self.vehicle is not None:

return False

return vehicle.can\_fit\_in\_spot(self)

def park\_vehicle(self, vehicle): # ...

def remove\_vehicle(self): # ...

Overwriting parking\_lot.py