Design Pattern:

1. Factory Method Pattern: This pattern can be used to create different types of users (Manager, Head of Department, Employee) based on the user type. Each user type has different privileges and pages associated with it. By using this pattern, creating objects according to the user type will become more flexible and scalable.

2. Template Method Pattern: This pattern can be used for designing the structure of the pages which have the same structure but different navigation bars and body contents. The navigation bar and body contents will vary based on the user type. By using this pattern, we can create a template method that defines the shell of an algorithm as a series of steps and allows the subclasses to provide the implementation for each step.

3. Composite Pattern: This pattern can be used to represent the hierarchical structure of the company. The composite pattern allows us to treat individual objects and compositions of objects uniformly.

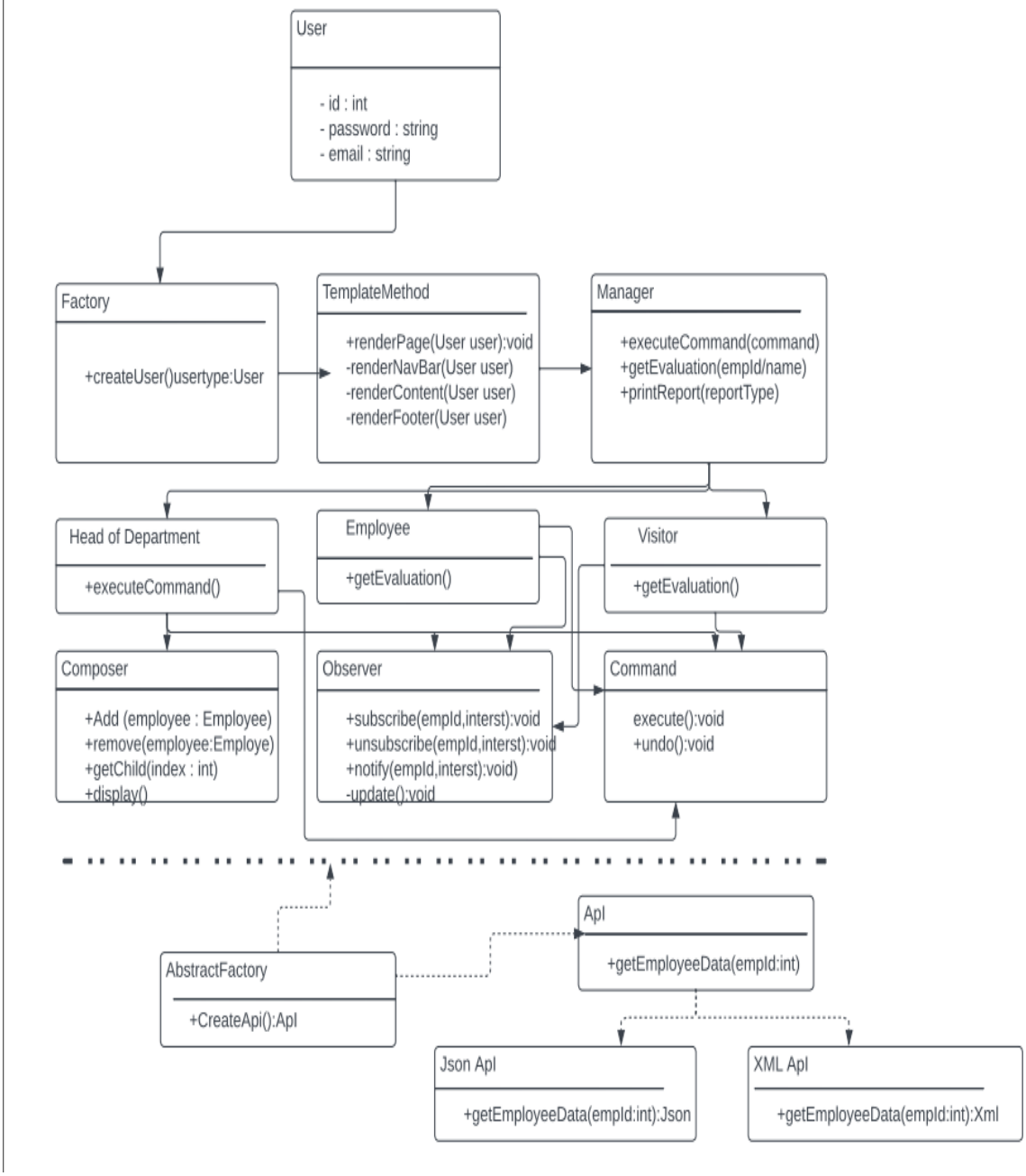
4. Observer Pattern: This pattern can be used to inform employees about new events related to their interests. The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

5. Command Pattern: This pattern can be used to execute operations such as hiring, promoting, firing, and making announcements by different users (Manager, Head of Department, Employee). The command pattern encapsulates a request as an object, thereby letting us parameterize clients with different requests, queue or log requests, and support undoable operations.

6. Facade Pattern: This pattern can be used to provide a simplified interface to a complex system. The facade pattern provides a higher-level interface that makes the system easier to use and understand. In this case, we can use this pattern to simplify the process of holding a meeting by providing a single method that encapsulates the necessary complex operations.

7. Abstract Factory Pattern: This pattern can be used to provide a common API for different types of mobile developers (JSON and XML). The abstract factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. In this case, we can use this pattern to create a common API that can be used by both JSON and XML developers.

Class Diagram:



Problem sets :

1 –

```python
def is_permutation_palindrome(s):
    # Convert the string to lowercase
    s = s.lower()

    # Create a hash table to store the frequency of each character
    freq = {}
    for c in s:
        if c in freq:
            freq[c] += 1
        else:
            freq[c] = 1

    # Count the number of characters with odd frequency
    num_odd_freq = 0
    for count in freq.values():
        if count % 2 != 0:
            num_odd_freq += 1

    # Return True if at most one character has odd frequency
    return num_odd_freq <= 1



input_str = "racecar"
output = is_permutation_palindrome(input_str)
print(output)  # Output: True
```

2-

```
    rows_with_zero = set()
    cols_with_zero = set()

    # iterate through matrix and check for zero elements
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == 0:
                rows_with_zero.add(i)
                cols_with_zero.add(j)

    # set all elements in rows with zero to zero
    for i in rows_with_zero:
        for j in range(len(matrix[0])):
            matrix[i][j] = 0

    # set all elements in columns with zero to zero
    for j in cols_with_zero:
        for i in range(len(matrix)):
            matrix[i][j] = 0

    return matrix

matrix = [
    [1, 2, 3],
    [4, 0, 6],
    [7, 8, 9]
]

print(len(matrix[0]))

output = set_zeroes(matrix)

print(output)

#output
# matrix = [
#   [1, 0, 3],
#   [0, 0, 0],
#   [7, 0, 9]
# ]
```

3-

```python
def count_ways(n):
    dp = [0]*(n+1)
    dp[0] = 1
    dp[1] = 1
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2] + dp[i-3]    # i - 1  --> بينقص اول خطوه
    return dp[n]



print(count_ways(5) , 'Ways')
# 13 ways
```

4-

```python
def permutations(s):
    """
    Returns a list of all permutations of a string s.
    """
    if len(s) == 1:
        return [s]

    result = []
    for i in range(len(s)):
        # Choose the i-th character as the first character
        # of the permutation and recursively generate all
        # permutations of the remaining characters
        first_char = s[i]
        remaining_chars = s[:i] + s[i+1:]
        sub_permutations = permutations(remaining_chars)

        # Add the first character to the front of each sub-permutation
        for sub_permutation in sub_permutations:
            result.append(first_char + sub_permutation)

    return result



output = permutations('abc')
```

```
print(output)
#['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

5-

```
def peaks_and_valleys(arr):
    # Sort the array in ascending order
    arr.sort()

    # Swap adjacent elements to form peaks and valleys
    for i in range(1, len(arr)-1, 2):
        arr[i], arr[i+1] = arr[i+1], arr[i]

    return arr


output = peaks_and_valleys([5, 3, 1, 2, 3])

print(output)
#[1, 3, 2, 5, 3]
```

-Mathematical Thinking.

- 1-To find the heavy bottle, you can follow the following steps:

1. Number the bottles from 1 to 20.

2. Take one pill from the first bottle, two pills from the second bottle, three pills from the third bottle and so on up to taking 19 pills from the 19th bottle. This way, the total number of pills you have will be 1+2+3+...+19 = 190.

3. Weigh the total number of pills using the scale. Let's say the weight is W grams.

4. The heavy bottle will be the bottle that contains the pills with a weight of 1.5 grams. To determine which bottle it is, we need to calculate the expected weight of the pills if all the bottles contained 1.0-gram pills.

5. The expected weight of the pills would be 190 grams since there are 190 pills in total. Therefore, if all the bottles contained 1.0-gram pills, the weight on the scale should be exactly 190 grams.

6. The difference between the actual weight (W) and the expected weight (190 grams) will tell us how much heavier the pills in the heavy bottle are compared to the other bottles.

7. Each extra pill in the heavy bottle adds an additional 0.5 grams to the total weight. Therefore, divide the difference between the actual weight and the expected weight by 0.5 to get the number of extra pills that the heavy bottle contains.

8. Divide the number of extra pills by the weight difference between 1.0-gram pills and 1.5-gram pills (0.5 grams) to get the bottle number of the heavy bottle.

For example, let's say the weight on the scale is 192 grams. The difference between the actual weight and the expected weight is 2 grams. Dividing this by 0.5 gives us the number of extra

pills, which is 4. Dividing 4 by 0.5 gives us 8, which is the bottle number of the heavy bottle. Therefore, the heavy bottle is the 8th bottle.

- 2-To solve this problem, we can use the principle of (inclusion-exclusion)

Now let's consider the case of three ants. There are three possible pairs of ants that can collide, as before. However, there is also the possibility of all three ants colliding at a single vertex. Each ant has two possible directions to choose from, so there are a total of 2*2*2=8 possible outcomes. Out of these, four outcomes result in a collision (one for each pair of ants and one for all three ants colliding at a vertex). Therefore, the probability of collision between three ants is 4/8 = 1/2.

- 3-
    - At the end of the 100 passes, only the lockers that have an odd number of factors will be open. Locker #100 will have been toggled on every pass from 1 to 100, so it will end up closed because it has an even number of factors: 1, 2, 4, 5, 10, 20, 25, 50, and 100.
    -
    - To find out which lockers are open at the end, we need to look for the lockers whose numbers have an odd number of factors. A locker with an odd number of factors must be a perfect square, since the factors come in pairs except when the number is the square of an integer (in which case one factor is repeated).
    -
    - There are 10 perfect squares less than or equal to 100: 1, 4, 9, 16, 25, 36, 49, 64, 81, and 100. Therefore, there are 10 lockers that will remain open at the end of the process.

-
- So the answer is: "There are 10 lockers open."

Image on 10 lockers only to understanding