CSC 6301 - Final Exam
Adam LaCasse

1. Include the docstrings documentation

```
class Queue:
    """
    A FIFO (First-In-First-Out) queue implementation using two internal
lists.

    This implementation uses two lists to achieve amortized O(1)
    enqueue and dequeue operations. Items are added to `a_in` and removed
    from `a_out`. When `a_out` is empty, all items from `a_in` are moved
    to `a_out` in reverse order, preserving FIFO semantics.

    Attributes:
        a_in (list): List for incoming enqueued elements.
        a_out (list): List for outgoing dequeued elements.
    """

    def __init__(self):
        """
        Initialize an empty queue with two internal lists.

        The `a_in` list holds newly enqueued items, while `a_out`
        holds items ready to be dequeued in FIFO order.
        """
        self.a_in = []
        self.a_out = []

    def enqueue(self, d):
        """
        Add an element to the back of the queue.

        Args:
            d: The data element to enqueue (any type).

        Time Complexity: O(1)
        """
        self.a_in.append(d)

    def dequeue(self):
        """
        Remove and return the element at the front of the queue.

        If `a_out` is empty, this method transfers all elements from
        `a_in` to `a_out` in reverse order to maintain FIFO behavior.
        This transfer happens at most once per batch of enqueued items,
        resulting in amortized O(1) performance.
```

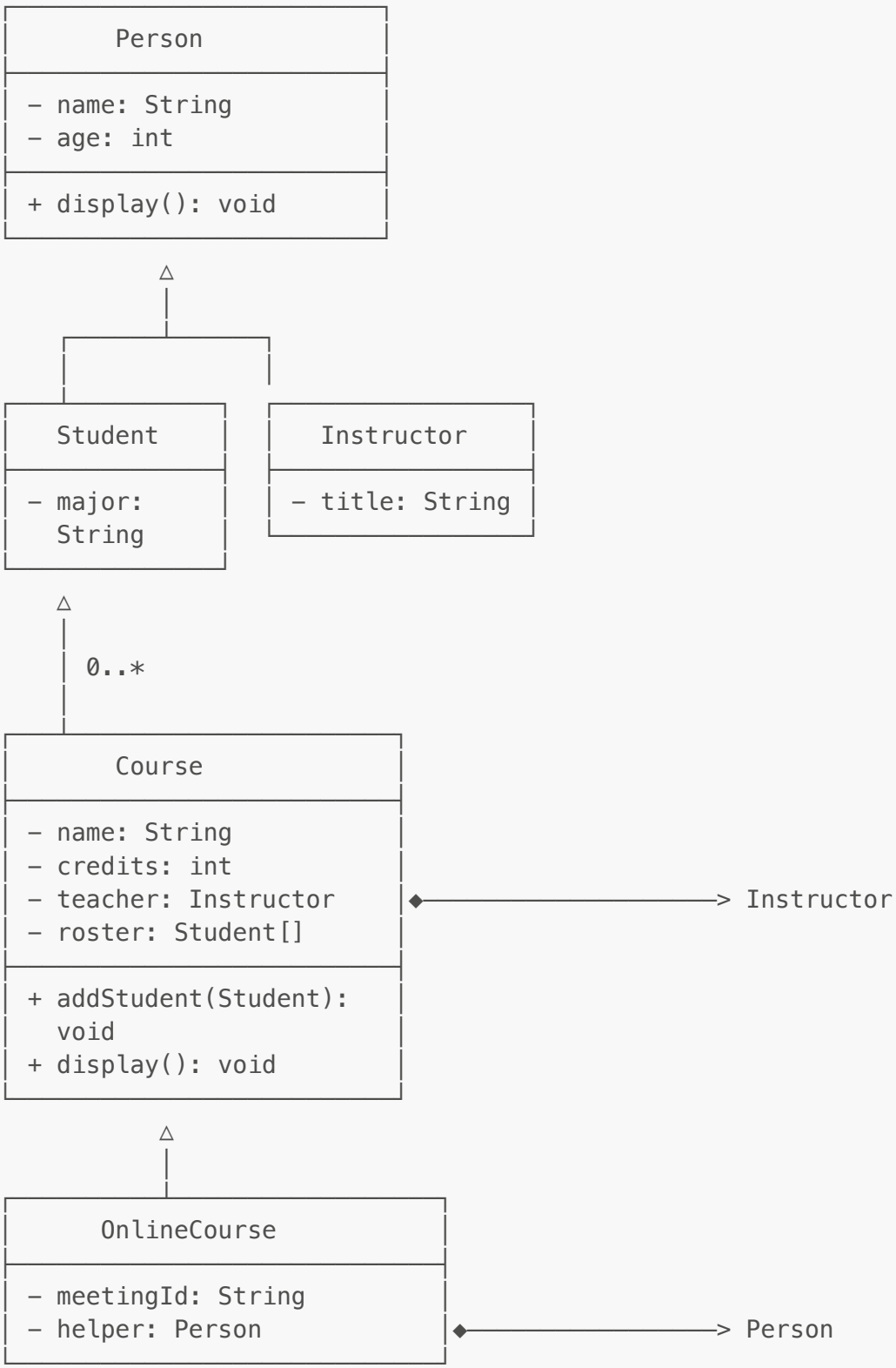
```
        The amortized analysis works because each element is transferred
        at most once from `a_in` to `a_out`, then dequeued once. Over n
        dequeue operations, the total cost is O(n) for transfers + O(n)
        for pops = O(2n), averaging to O(1) per operation.

        Returns:
            The front element of the queue.

        Raises:
            IndexError: If the queue is empty when dequeue is called.

        Time Complexity: Amortized O(1), worst-case O(n) when
transferring.
        """
        if (self.a_out == []):
            for d in self.a_in:
                self.a_out.append(d)
            self.a_in = []
        return self.a_out.pop(0)
```

2. Convert from Python to Java

```
import java.util.Scanner;
import java.util.Set;

public final class RockPaperScissors {

    private static final Set<String> VALID_CHOICES = Set.of("r", "p",
"s");

    public static void main(String[] args) {
        try (Scanner scanner = new Scanner(System.in)) {
            play(scanner);
        }
    }

    private static void play(Scanner scanner) {
        String playerChoice;
        while (true) {
            System.out.print("Enter (r)ock, (s)cissors, or (p)aper: ");
            playerChoice = scanner.nextLine().trim().toLowerCase();
            if (!VALID_CHOICES.contains(playerChoice)) {
                System.out.println("Only 'r', 's', or 'p' are valid
inputs! Please try again.");
            } else {
                break;
            }
        }

        String computerChoice = pickComputerChoice();
        announceOutcome(playerChoice, computerChoice);
    }

    private static String pickComputerChoice() {
        double rand = Math.random();
        if (rand < 1.0 / 3.0) {
            return "r";
        } else if (rand < 2.0 / 3.0) {
            return "s";
        }
        return "p";
    }

    private static void announceOutcome(String playerChoice, String
computerChoice) {
        if (playerChoice.equals(computerChoice)) {
            System.out.println("It is a tie!");
        } else if (playerLoses(playerChoice, computerChoice)) {
            System.out.printf("Sorry, you lost as I had %s.%n",
describeChoice(computerChoice));
        } else {
            System.out.printf("Congrats, you won as I had %s.%n",
```

```
describeChoice(computerChoice));
        }
    }

    private static boolean playerLoses(String playerChoice, String
computerChoice) {
        return ("r".equals(playerChoice) && "p".equals(computerChoice))
            || ("p".equals(playerChoice) && "s".equals(computerChoice))
            || ("s".equals(playerChoice) && "r".equals(computerChoice));
    }

    private static String describeChoice(String choice) {
        return switch (choice) {
            case "r" -> "rock";
            case "p" -> "paper";
            case "s" -> "scissors";
            default -> throw new IllegalArgumentException("Unexpected
choice: " + choice);
        };
    }
}
```

3. Class Diagram



4. Java Collections

Stack is a legacy LIFO class that extends Vector, so push/pop/peek operate on the synchronized top element inherited from Vector. Queue is a FIFO interface in `java.util`, typically implemented by LinkedList, PriorityQueue, or ArrayDeque, exposing offer/poll/peek while letting each class decide on synchronization. Because Stack carries Vector's synchronization overhead, it is slower and less flexible than most queue/deque implementations. Modern Java therefore prefers Deque implementations such as ArrayDeque for both stack-like and queue-like usage instead of the old Stack class.

5. SDLC

Separating development and testing teams provides independent validation, reducing confirmation bias where developers might overlook their own mistakes. Testers bring a fresh perspective focused on breaking the code and uncovering edge cases that developers may not anticipate. This division of responsibilities ensures higher quality assurance, clearer accountability, and better adherence to requirements. It also allows each team to specialize - developers optimize for implementation efficiency, while testers focus on coverage, usability, and defect detection - ultimately delivering more robust and reliable software.

6. Version Control

To recreate a reusable environment, use a virtual environment and a `requirements.txt` file. First, create and activate the environment. Then, install the packages and save them to a file:

```
python -m venv venv
source venv/bin/activate  # for macOS/Linux; on Windows:
venv\Scripts\activate
pip install numpy==1.18.5 xlwt==1.3.0
pip freeze > requirements.txt
```

This creates a `requirements.txt` file, which is the additional file needed. Others can then perfectly recreate the environment by running `pip install -r requirements.txt`. No changes are needed in the `.py` files themselves.

7. Profiling

The function that should be optimized is `check(mat)`. Profiling shows it dominates execution time, being called tens of millions of times and accounting for the largest share of total runtime. This function performs nested loops over the matrix, giving it O(n^2) complexity per call, and it is invoked repeatedly within `layer()`. The inner loops count occurrences of 0s and 1s, then check balance conditions—calculations that could potentially be memoized or restructured to avoid redundant matrix scans. Optimizing this part (e.g., precomputing partial sums, caching results, or using vectorized operations) would significantly reduce total execution time.

8. SDM

Lean is most similar to Waterfall: both can operate as structured, phase-oriented flows, and Lean optimizes that flow (waste reduction, value-stream discipline) without requiring short iterations or CI/CD. Agile and DevOps explicitly break phase gates via iteration and continuous delivery. A fair alternate lens says Agile "looks" similar because each sprint contains plan-build-test mini-cycles, but that's procedural resemblance; structurally Agile rejects linear phases — hence Lean remains closer.