Smart Canteen Queue Management & Waiting Time Prediction System

Project Report

**Submitted By: ANJALI SONKAR
**Course: Problem Solving And programming
**Name of Institution: VIT Bhopal

Date: 24/11/2025

-

Table of Contents

Introduction

Overview This lab will introduce participants to the various concepts surrounding electronic communications. Topics covered include electronic signals, devices, and a wide range of communication systems.

The **Smart Canteen Queue Management & Waiting Time Prediction System** is a console-based application designed to solve the real-world problem of inefficient queue management in college canteens. The issues of students at lunchtime are long queues, uncertainty over the time spent waiting, and no reliable information about seat availability. This system provides real-time queue status, waiting time predictions, seat availability tracking, and intelligent counter recommendations.

1.2 Purpose

The purpose of this project is to:

-Apply programming concepts learned within the course to solve a practical problem.

- Knowing object-oriented programming, data structure, and algorithm basics

- Demonstrate competence in software design, modularization, and testing

1.3 Scope

This project focuses on the following:

- Waiting time estimation depending on the order's complexity

- Providing the best guidance for counter selection

The project does not include:

- Payment Processing

- Database persistence

- Web or mobile interface

- User authentication

- Inventory management

---

2. Problem Statement

2.1 Real-World Problem

College canteens face a big problem in peak time:

1. **Long Queues**: Several counters have queues of varying lengths, and students don't know which one is the fastest.

2. **Uncertain Waiting Time**: Students have no idea how long they need to wait before getting served

6. **Crowd Chaos**: Without proper management, canteens become chaotic during peak hours.
2.2 Impact
- **Students**: Waste time, miss classes, experience frustration
Canteen Staff: Inability to handle crowds, delay in serving

2.3 Solution Approach

This system provides:

Real-time queue status for all counters

- Seat availability tracking

---

3. Functional Requirements
3.1 Queue Management Module
Requirement FR1: The system shall manage multiple food counters (minimum 3 counters)

- Customer name

- Order items (support for multiple items)
- Automatic assignment to best counter
FR3 Requirement: The system shall permit serving customers from queue, first in first out.

FR4: The system shall show real-time queue length for every counter
Implement:
- `Counter` class handles counter queues individually by using a `deque` data structure.

- The `QueueSystem` class coordinates several counters
Operations on queues : `add*customer()`, `serve*next()`, `finish_serving()`
3.2 Waiting Time Prediction Module

FR5: The system shall calculate serving time based on order complexity

FR6: The system shall estimate total waiting time for new customers considering:

- Number of customers ahead in queue

- Complexity of each order

- Remaining time still serving a customer

FR7: The system shall use a dynamic algorithm that considers
- Quantity ordered
- Item types (combos take longer, beverages are quick)

- Base serving time + complexity multiplier

Implementation:

- `Customer` class calculates the complexity score based on items

- Formula: `Base Time (30s) + (Complexity Score × 10s)`
3.3 Seat Availability Module
FR8 Requirement: The system shall count the seats in total of 50.

FR9: The system shall keep track of occupied and vacant seats in real-time

FR11: The system shall automatically assign a seat when a customer is being served.

- `SeatManager` class will track seat status

- automatic seat allocation in `serve_customer()` method

FR12: The system shall compare wait times across all counters

**FR13: The system shall recommend counter with shortest wait time.

FR14 Requirement: The system shall show comparison with other counters
Implementation:

- Compares `get*estimated*wait_time()` for all counters

- Returns counter with minimum wait time

3.5 User Interface Module

FR15 requirement: The system shall provide a clear menu-driven interface

FR16: The system shall display informative status information
FR17: The system shall handle validation of user input
Implementation:

- `DisplayManager` class handles all UI operations

Menu options: Add Customer, Serve Customer, View Status, View Seats, Get Recommendation, Exit

FR19: The system shall track errors and warnings
Implementation:

- Logs : INFO, ERROR, WARNING levels

- Timestamped entries

PSI ---
4. Non-Functional Requirements

4.1 Performance
Requirement NFR1: Queue operations shall be efficient

Implementation:

- Linear search for best counter (acceptable for 3 counters)

- No unnecessary computations

Measurements: Queue operations complete in < 1ms

4.2 Usability

- Clear menu with numbered options

- Descriptive keys and messages

- Helpful error messages

Measurement: Users can use the system without training
4.3 Error Handling
Requirement NFR3: System shall handle errors gracefully
Implementation:
- Input validation: type checking, range validation

- Try-except blocks for file operations
- User-friendly error messages
System continues operation after errors
Examples:

- Invalid counter selection. Please check and try again.

- File I/O errors !’ Fallback to console output

Implementation:

- Queue operations are atomic

- Seat count always correct

- Customer IDs are unique and in sequence
Measurement: No data loss or corruption was observed
4.5 Maintainability
Requirement NFR5: Code shall be well-organized and documented

Implementation:

Modular design: separate files for different concerns

- Clear class and method names

- Consistent code style
- Separation of concerns, UI, logic, utilities

- `main.py`: Entry point

- `display.py`: UI layer
- `logger.py`: Logging
- `utils/`: Helper functions

Requirement NFR6: System shall log operations for debugging
Implementation:

Timestamped entries

- Log file in `logs/` directory
4.7 Resource Efficiency

- In-memory data structures, no database overhead

- Efficient algorithms: O(n) for searches

- No external dependencies (only Python standard library)

5. Architecture of the System

5.1 High-Level Architecture

The system follows a **layered architecture**:

% Application Layer %

% (main.py, display.py, logger.py) %

ÇN-Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

%

%¼

% Business Logic Layer %

% - Customer, Counter, SeatManager %

% - QueueSystem %

% %

%

%¼

% Utility Layer %

```

- `main.py`: The entry point, orchestrates all components

- `display.py`: User interface and display logic
```

- `queue_system.py`: Core functionality
- `Customer`: Represents a customer with order
- `Counter`: Manages individual counter queue

- `QueueSystem`: Master system coordinator

3. **Utility Layer**:

- `utils/utils.py`: Helper functions (formatting, screen clearing)

1. User interacts with `main.py`

2. `main.py` calls methods of `QueueSystem`
3. `QueueSystem` updates data structures: queues, seats
4. `DisplayManager` retrieves and formats data

5. Information displayed to user

6. The `Logger` records the operations

- **Object-oriented design**: Classes encapsulate related functionality

- **Modularization**: Different concerns in different files

- **Composition**: QueueSystem contains Counters and SeatManager

- **Top-Down Design**: High-level functions call lower-level functions

-

6. Diagrams of Designs
6.1 Use Case Diagram
Actors: Student, Staff, Administrator

Use Cases:

- View Queue Status

- Get Best Line Recommendation
- Check Seat Availability
- Add Customer to Queue
- Serve Customer from Queue

- View System Logs

(See DESIGN_DIAGRAMS.md for detailed diagrams)

6.2 Workflow Diagram
It shows the main flow of the program:
1. System initialization
2. Show main menu

5. Back to menu

6.3 Sequence Diagram

1. **Add Customer Flow**: Student !’ Main !’ QueueSystem !’ Counter

2. **Serve Customer Flow**: Staff !’ Main !’ QueueSystem !’ Counter !’ SeatManager

6.4 Class Diagram
it gives all classes and their relationships:

- Customer !’ Queue !’ Counter

(All diagrams are available in DESIGN_DIAGRAMS.md)

---

7. Design Decisions & Rationale

7.1 Data Structure Choice: deque vs list
Decision: Use `collections.deque` for queue implementation

- `deque.popleft()` is O(1) vs `list.pop(0)` which is O(n)

Better performance for queue operations

- Standard python library, no external dependency

7.2 In-Memory vs Database Storage

Decision: Use in-memory data structures
Rationale:

- Can be extended to database later

Decision: console-based interface
Rationale:

- Works on all platforms
- Focus on the core logic rather than UI framework

7.4 Multiple Counters vs Single Queue
Decision: Use multiple counters (3 counters)

7.5 Dynamic Serving Time Algorithm

- More accurate than simple queue counting
- Demonstrates algorithm design

- Realistic approach: complex orders take longer

7.6 Modular Design

- Easier to understand and maintain
- Follows principles of software engineering

7.7 Class-Based Design

Rationale:

- Demonstrates OOP concepts
- Easier to extend and modify

- Industry-standard approach

-

8. Implementation Details

8.1 Technology Stack
- **Programming Language**: Python 3.x
- **Libraries Used:

- **No External Dependencies**: Only uses Python standard library
8.2 Key Algorithms
#### 8.2.1 Complexity Calculation Algorithm

```python

def *calculate*complexity(self):
```

for item in self.items:

complexity += 2
elif 'beverage' in item.lower():

elif 'dessert' in item.lower():
complexity += 1
return complexity
```

#### 8.2.2 Serving Time Estimation

def get*estimated*serving_time(self):

time*per*complexity = 10
return base*time + (self.complexity*score * time*per*complexity)

```

8.2.3 Calculation of the Waiting Time
```python
def get*estimated*wait_time(self):

total_time = 0

if self.serving_customer:
elapsed = (datetime.now() - self.serving*start*time).total_seconds()

if remaining > 0:

# Add time for all customers in queue

```
Time Complexity: O(n) where n = length of queue
#### 8.2.4 Best Counter Selection

best_counter = self.counters[0]

for counter in self.counters[1:]:

min*wait*time = wait_time

return best_counter

```counter```

8.3 File Structure

smart_canteen/

% % %main.py   # Entry point (70 lines)

% % %queue_system.py   # Core logic (403 lines)

% % %display.py   # UI management (145 lines)

% % %test*queue*system.py   # Tests (121 lines)

% % %init.py

% % %statement.md
% % %DESIGN_DIAGRAMS.md
% % %PROJECT_REPORT.md

% % %screenshots/

```

**Total Lines of Code**: ~866 lines excluding comments and documentation

#### Customer Class
- `init()`: Initialize customer with ID, name, items

#### Class Counter
- `add_customer()`: Add customer to queue

- `get*queue*length()`: Get number of waiting customers

- `get*estimated*wait_time()`: Calculate total wait time

- `occupy_seat()`: Allocate seats

- `free_seat()`: Reliberar assentos
- `get*vacant*seats()`: Get available seats
- `get*occupancy*percentage()`: Calculate occupancy

- `add*customer*to_queue()`: Interactive customer addition

- `get*all*queue_status()`: Get status of all counters
- `get*best*line_recommendation()`: Recommend best counter

8.5 Input/Output Structure

**Input**:

- Customer name (string)

- Selection of menu items, separated by commas
```

- Counter selection No. 1-3
- Menu choice (number 1-6)

Output:

- Queue status (queue length, wait times)
- Seat availability (occupied, free, in %)

8.6 Error Handling

- **Input Validation**: Type checking, range validation

- **File Operations**: Try-except for writing in log file
- **Queue Operations**: Check for empty queue before serving
- **Seat Operations**: Check for availability before allocation

- **Friendly Messages**: unmistakable descriptions of errors

---

9. Screenshots / Results

9.1 Running a Program

- Clear formatting and borders

Screenshot 2: Adding Customer

- Customer name input
- Menu item selection
- automatic counter assignment

Display serving time and complexity:

Screenshot 3: Queue status
- All counters that have queue lengths
Estimated time spent waiting

- Currently serving customers

Screenshot 4: Seat Availability
- Total, occupied, vacant seats

- Occupancy percentage

Visual representation as bars

Screenshot 5: Best Line Recommendation

- Recommended counter

- Comparisons with other counters

Wait time differences

Screenshot 6: Test Results

- Test output showing success

9.2 Test Results

```
============================================================
RUNNING VALIDATIONTESTS
============================================================

  @ > 2 5 @0A0> 7 4 0 =7 8 5 0 7 G 2 K 0 ? > ; = 5 = 0 !
Testing Counter Operations.


Testing Seat Manager.



Testing Wait Time Calculation.

Testing Queue System.
Queue system initialization test: passed!

============================================================

ALL TESTS PASSED!

============================================
```

9.3 Sample Execution Flow
1. **Start Program**: System initializes 3 counters and 50 seats

3. **View Status**: View queue lengths and waiting times

4. **Serve Customers**: Serve customers from counters
5. **Check Seats**: Display seat availability after serving
6. **Get Recommendation**: System recommends best counter

10.1 Testing Strategy

Unit Testing: Test individual components
Creation and calculation of customer complexity
- Counter queue operations
- Seat management

Wait time calculations

Integration Testing: Test interactions among components

- Initialization of the queue system

- Customer addition and serving flow
- Seat allocation after serving

Validation Testing: Test system correctness

- All tests in the test*queue*system.py
- Assertions check expected behaviour

10.2 Test Cases

- **Input**: Customer ID=1, Name="Test Student", Items=["Rice & Curry", "Beverage"]
- **Expected**: Customer created with complexity > 0

- **Result**: Pass

#### Test 2: Counter Operations
- **Input:** Add 2 customers to counter

- **Expected**: Queue length = 2, FIFO serving

- **Result**: Pass

- **Expected**: Empty seats = 45
- **Result**: Pass
#### Test 4: Wait Time Calculation

- **Input**: 2 customers in queue having different complexities

#### Test 5: System Initialization

- **Result**: Pass

Manual Testing
Scenario 1: Adding several customers
- Add 5 customers with different orders

Check wait times are calculated right

- Check seats are assigned
- Check queue updates correctly

(x)Server from empty queue (exception handling)

Customer class: Creation, complexity, serving time

- Counter class: Add, serve, queue length, wait time

11. Challenges Encountered

11.1 Challenge 1: Wait Time Calculation Accuracy

Solution:
Serving start time for a current customer

Queue: SUM all serving times of customers

11.2 Challenge 2: Best Counter Selection

Problem: Need to compare wait times across all counters and select minimum

- Compare `get*estimated*wait_time()` of each counter

11.3 Challenge 3: Code Organization

Problem: How to manage code so it's easier to read and maintain

Solution:

- Separate files by concern: logic, UI, utilities
- Use classes to encapsulate related functionality

Problem: Make the console interface clear and user-friendly

Solution:

11.5 Challenge 5: Testing

Solution:
- Create test file with test functions

- Unit test components, integration

- Learned to design classes with appropriate attributes and methods

- Understood encapsulation and data hiding
- Practiced class composition and relationships

- Used `deque` for efficient queue operations

- Understood time complexity of different operations

3. **Algorithms**:

Implemented searching algorithm (finding best counter)

- Designed calculation algorithms: waiting time, complexity

- Practiced modularization and separation of concerns

12.2 Problem-Solving Skills

- Broke down into smaller sub-problems

2. **Algorithm Design**:

Created wait time estimation algorithm

3. **Debugging**:

1. **Documentation**:

- Learned importance of code comments

Created comprehensive README and documentation.

- Documented design decisions

2. **Testing**:

- Wrote validation tests
- Learned to verify system correctness

3. **Code Quality**:

- Practiced clean code principles

- Created useful tool for canteen management
Demonstrated real-world application of programming
2. **User Experience**:

- Developed intuitive interface

- **Modular Design**: Breaking code into modules is a way of making code more understandable and maintainable.