System Requirements Specification Index

For

Flower Shop Inventory Management System

Version 1.0

IIHT Pvt. Ltd.

fullstack@iiht.com

TABLE OF CONTENTS

Flower Shop Inventory Management System

System Requirements Specification

1 PROJECT ABSTRACT

"Bloom & Blossom" is a rapidly expanding flower shop chain with 15 locations across the region. As the business grows, their manual inventory system has become inefficient and error-prone, leading to issues like overordering perishable stock, unexpected shortages during peak seasons, and financial losses from discarded expired flowers. Management needs a robust inventory system that handles the unique challenges of their perishable products while maintaining reliable operation even when faced with unexpected inputs, network issues, or user errors. The Flower Shop Inventory Management System will address these challenges with comprehensive error and exception handling throughout all critical business operations.

2 BUSINESS REQUIREMENTS:

Screen Name	Console input screen
Problem Statement	1. Handle inventory management for various flower types 2. Process customer orders with proper validation 3. Track daily sales and generate reports 4. Implement robust error handling using all four exception handling blocks 5. Maintain data integrity during exceptional conditions

3 CONSTRAINTS

3.1 CLASS AND METHOD REQUIREMENTS

1. `FlowerShopException` Class:
   - Base exception for the entire system
   - Methods
     - § `__init__(message, error_code=None)`: Initialize with message and optional error code
     - § `__str__()`: Return formatted error message with error code if available
2. Required Exception Subclasses:
   - `InvalidFlowerDataError`: For flower data validation errors
   - `InvalidOrderError(reason)`: For order-related errors with constructor taking reason parameter
   - `OutOfStockError(flower_name, requested, available)`: For insufficient inventory
   - `ExpiredFlowerError(flower_name, expiry_date)`: For expired flowers
3. `Flower` Class
   - Attributes:
     - § `name`: String name of the flower
     - § `price`: Float price per stem
     - § `quantity`: Integer number of stems
     - § `freshness_date`: Datetime when flower expires
   - Methods:
     - § `__init__(name, price, quantity, freshness_days=7)`: Initialize flower with try-except-else pattern
     - § `validate_name(name)`: Validate name format and raise appropriate exception

- § `validate_price(price)`: Validate price is positive number and raise appropriate exception
- § `validate_quantity(quantity)`: Validate quantity is non-negative integer and raise appropriate exception
- § `is_fresh()`: Return True if flower is still fresh, False otherwise
- § `__str__()`: Return string representation of flower

4. `Inventory` Class
   - Attributes:
     - § `flowers`: Dictionary mapping flower names to Flower objects
     - § `transaction_log`: List of dictionaries containing transaction records
   - Methods:
     - § `__init__()`: Initialize empty inventory
     - § `add_flower(flower)`: Add flower to inventory with full try-except-else-finally pattern
     - § `remove_flower(flower_name, quantity)`: Remove flowers from inventory with full try-except-else-finally pattern
     - § `check_stock(flower_name)`: Check stock level with try-except-else pattern
     - § `get_all_flowers()`: Return list of all flowers in inventory
     - § `get_transaction_log()`: Return the transaction log
5. `Order` Class
   - Attributes:
     - § `customer_name`: Name of the customer
     - § `inventory`: Reference to Inventory object
     - § `items`: Dictionary mapping flower names to quantities
     - § `status`: String status of order ("new", "processed", or "failed")

§ `total_price`: Float total price of the order

- Methods:
  - § `__init__()`: Initialize empty inventory
  - § `__init__(customer_name, inventory)`: Initialize with try-except-else pattern
  - § `add_item(flower_name, quantity)`: Add item with full try-except-else-finally pattern
  - § `process()`: Process order with full try-except-else-finally pattern
  - § `_rollback_inventory(processed_items)`: Roll back inventory changes with try-except-finally pattern
  - § `__str__()`: Return string representation of order

3.2 Function Constraints

1. `generate_daily_report(inventory)`:
   - Generate sales and inventory report with full try-except-else-finally pattern
   - Return dictionary with report data
   - Must use all four exception handling components
2. `main()`:
   - Demonstrate all exception handling patterns
   - Must include examples of all error types
   - Must show proper usage of try-except-else-finally blocks

3.3 Error Handling Constraints

1. `try` Block Usage:
   - All operations that could raise exceptions must be in try blocks
   - Code should be structured to isolate potential error sources
   - Required in all methods that modify state or access external resources
2. `except` Block Usage:

- Must catch specific exceptions, not generic Exception where possible
- Handle exceptions appropriately with meaningful error messages
- Re-raise exceptions when appropriate
- Must include proper error logging

3. `else` Block Usage:
   - Must be used for code that should run only when no exceptions occur
   - Must implement logical separation between error-prone and safe code
   - Use for transaction completion and state updates
   - Required in all methods with try-except blocks

4. `finally` Block Usage:
   - Must be used for cleanup operations that always need to happen
   - Must implement resource management and logging
   - Handle rollback operations when required
   - Required in all methods with state changes or resource use

3.4 Data Constraints

1. Transaction Log Structure:
   - Each entry must be a dictionary with:
     - `type`: String type of transaction (e.g., "add", "remove")
     - `flower_name`: String name of the flower
     - `quantity`: Integer quantity affected
     - `status`: String status ("pending", "completed", "failed")
     - `error`: String error message if failed (optional)

2. Order Structure:

- Must track customer information
- Must track all items and quantities
- Must track order status
- Must track total price

3. Report Structure:
    - Must include date
    - Must include inventory statistics
    - Must include transaction summary
    - Must include stock alerts for low inventory

3.5 Constructor/Destructor Constraints

1. No bare except blocks allowed
2. All transaction operations must use the complete try-except-else-finally pattern
3. Logical flow control must not rely on exceptions (exceptions for exceptional cases only)
4. All resource cleanup must be in finally blocks
5. All transactions must have rollback capability in case of errors

4. Template code Structure:

1. Exception Classes:
    - Base FlowerShopException
    - Specialized exception subclasses

2. Flower Class:
    - Attributes for flower properties
    - Validation methods
    - Error handling with try-except-else

3. Inventory Classes:

- Flower storage
- Transaction log
- Operations with full exception handling

4. Order Class:

- Customer information
- Order items
- Processing with transaction integrity

5. Utility Functions:

- Report generation
- Helper functions

4. Main Program:

- Demonstration of all exception handling patterns

5. DETAILED FUNCTION IMPLEMENTATION GUIDE

5.1 Exception Classes Implementation

1. **Write a Python exception class to serve as base for all flower shop exceptions.**
   Define: `FlowerShopException(Exception)`

   The class should:

   - Accept `message` (string) and optional `error_code` (string) parameters in constructor
   - Store both message and error_code as instance attributes
   - Override `__str__()` method to return formatted string
   - If error_code is provided, format as: `"[{error_code}] {message}"`
   - If no error_code, return just the message

- Example: `FlowerShopException("Test error", "E001")` returns `"[E001] Test error"`

2. **Write a Python exception class for invalid flower data errors.**

   Define: `InvalidFlowerDataError(FlowerShopException)`

   The class should:

   - Inherit from FlowerShopException
   - Pass message and error_code to parent constructor
   - Used for validation failures in flower creation
   - Example usage: `raise InvalidFlowerDataError("Invalid price", "F001")`

3. **Write a Python exception class for invalid order errors.**

   Define: `InvalidOrderError(FlowerShopException)`

   The class should:

   - Inherit from FlowerShopException
   - Accept `reason` parameter in constructor
   - Format message as: `"Invalid order: {reason}"`
   - Always use error code "E001"
   - Example: `InvalidOrderError("Empty order")` creates message "Invalid order: Empty order"

4. **Write a Python exception class for out of stock errors.**

   Define: `OutOfStockError(FlowerShopException)`

   The class should:

   - Inherit from FlowerShopException
   - Accept `flower_name`, `requested`, and `available` parameters
   - Format message as: `"Insufficient stock for {flower_name}. Requested: {requested}, Available: {available}."`
   - Always use error code "I001"
   - Example: `OutOfStockError("Rose", 50, 30)` for insufficient Rose stock

5. **Write a Python exception class for expired flower errors.**

   Define: `ExpiredFlowerError(FlowerShopException)`

   The class should:

   - Inherit from FlowerShopException

- Accept `flower_name` and `expiry_date` parameters
- Format message as: `"Flower '{flower_name}' has expired. Freshness date: {expiry_date}"`
- Always use error code "I002"
- Example: `ExpiredFlowerError("Lily", "2024-12-01")` for expired Lily

5.2 Flower Class Implementation

6. **Write a Python class constructor to initialize a flower with validation.**

   Define: `Flower.__init__(self, name, price, quantity, freshness_days=7)`

   The method should:

   - Use try-except-else pattern for validation
   - In try block: call `validate_name()`, `validate_price()`, `validate_quantity()`
   - In except block: catch `(ValueError, TypeError)` and raise `InvalidFlowerDataError` with chaining
   - In else block: set instance attributes (`name`, `price`, `quantity`, `freshness_date`)
   - Convert price to float and quantity to integer
   - Calculate `freshness_date` as current datetime + timedelta(days=freshness_days)
   - Default freshness_days is 7 if not provided
7. **Write a Python function to validate flower name format.**

   Define: `validate_name(self, name)`

   The method should:

   - Check if name is string and not empty after stripping whitespace
   - Raise `InvalidFlowerDataError("Flower name cannot be empty", "F002")` for empty names
   - Use regex pattern `r'^[a-zA-Z\s\-\']+$'` to validate format
   - Allow only letters, spaces, hyphens, and apostrophes
   - Raise `InvalidFlowerDataError` with code "F003" for invalid format
   - Include the invalid name in error message for debugging
8. **Write a Python function to validate flower price.**

   Define: `validate_price(self, price)`

The method should:

- Use try-except pattern to convert price to float
- Check if converted price is positive (> 0)
- Raise `InvalidFlowerDataError("Price must be positive", "F004")` for non-positive prices
- In except block: catch `(ValueError, TypeError)` and raise `InvalidFlowerDataError` with code "F005"
- Chain the original exception using `from e`
- Include the invalid price value in error message

9. **Write a Python function to validate flower quantity.**

   Define: `validate_quantity(self, quantity)`

   The method should:

   - Use try-except pattern to convert quantity to integer
   - Check if converted quantity is non-negative (>= 0)
   - Raise `InvalidFlowerDataError("Quantity cannot be negative", "F006")` for negative quantities
   - In except block: catch `(ValueError, TypeError)` and raise `InvalidFlowerDataError` with code "F007"
   - Chain the original exception using `from e`
   - Include the invalid quantity value in error message

10. **Write a Python function to check if flower is still fresh.**

    Define: `is_fresh(self)`

    The method should:

    - Compare current date with freshness_date
    - Use `datetime.now().date() <= self.freshness_date.date()`
    - Return True if flower is still fresh (including today)
    - Return False if flower has expired
    - Include current day as fresh (use <= not <)

11. **Write a Python function to provide string representation of flower.**

    Define: `__str__(self)`

    The method should:

    - Return formatted string with all flower information
    - Format: `"{name}: ${price:.2f}, {quantity} in stock, fresh until {date}"`

- Use `.strftime('%Y-%m-%d')` for date formatting
- Example: `"Rose: $4.99, 50 in stock, fresh until 2024-12-15"`

5.3 Inventory Class Implementation

12. **Write a Python class constructor to initialize empty inventory.**

    Define: `Inventory.__init__(self)`

    The method should:

    - Initialize `flowers` as empty dictionary (flower_name: Flower object)
    - Initialize `transaction_log` as empty list
    - Both attributes should be instance variables
13. **Write a Python function to add flowers to inventory with transaction logging.**
    Define: `add_flower(self, flower)`

    The method should:

    - Use try-except-else-finally pattern
    - Create transaction dictionary with fields: `type`, `flower_name`, `quantity`, `status`
    - Set initial status as 'pending'
    - In try block: validate flower is Flower instance, handle duplicate flowers by adding quantities
    - In except block: catch FlowerShopException, set transaction status to 'failed', add error field
    - In else block: set transaction status to 'completed', return True
    - In finally block: append transaction to log, print transaction result
    - For duplicates: add new quantity to existing flower's quantity
14. **Write a Python function to remove flowers from inventory with validation.**

    Define: `remove_flower(self, flower_name, quantity)`

    The method should:

    - Use try-except-else-finally pattern with rollback capability
    - Create transaction dictionary and track `inventory_updated` flag
    - Validate flower_name is non-empty string and quantity is positive integer
    - Check flower exists in inventory, is fresh using `is_fresh()`, and has sufficient stock
    - Update inventory by subtracting quantity from flower's quantity
    - In except block: rollback changes if inventory was modified, log failed transaction

- In else block: set transaction status to 'completed', return True
- In finally block: append transaction to log
- Raise appropriate exceptions: `InvalidFlowerDataError`, `FlowerShopException`, `ExpiredFlowerError`, `OutOfStockError`

15. **Write a Python function to check stock level for specific flower.**

    Define: `check_stock(self, flower_name)`

    The method should:

    - Use try-except-else pattern
    - In try block: check if flower_name exists in flowers dictionary
    - Raise `FlowerShopException(f"Flower not found: {flower_name}", "I008")` if not found
    - In except block: re-raise the FlowerShopException
    - In else block: return the flower's quantity
    - No modification of inventory state

16. **Write a Python function to get all flowers in inventory.**

    Define: `get_all_flowers(self)`

    The method should:

    - Return list of all Flower objects in inventory
    - Use `list(self.flowers.values())`
    - Return empty list if no flowers in inventory

17. **Write a Python function to get transaction log.**

    Define: `get_transaction_log(self)`

    The method should:

    - Return the complete transaction_log list
    - No modification of the log
    - Return copy if needed to prevent external modification

5.4 Order Class Implementation

18. **Write a Python class constructor to initialize customer order.**

    Define: `Order.__init__(self, customer_name, inventory`

    The method should:

- Use try-except-else pattern for validation
- Validate customer_name is non-empty string
- Validate inventory is Inventory instance
- Raise `InvalidOrderError("Customer name cannot be empty")` for invalid names
- Raise `FlowerShopException("Invalid inventory object", "0001")` for invalid inventory
- In else block: initialize `customer_name`, `inventory`, `items` (empty dict), `status` ("new"), `total_price` (0.0)

19. **Write a Python function to add items to order with validation.** Define: `add_item(self, flower_name, quantity)` The method should:
    - Use try-except-else-finally pattern
    - Check order status is "new" (cannot modify processed orders)
    - Validate flower_name is non-empty string and quantity is positive integer
    - Check flower exists in inventory and is fresh
    - Calculate available quantity considering current order quantities
    - Check sufficient stock is available
    - In except block: re-raise FlowerShopException, catch other exceptions and wrap
    - In else block: update items dictionary, calculate and update total_price, return total_price
    - In finally block: log the attempt
    - For duplicate items: add to existing quantity in order
20. **Write a Python function to process order and update inventory.**

    Define: `process(self)`

    The method should:

    - Use try-except-else-finally pattern with rollback capability
    - Track `processed_items` list for rollback purposes
    - Validate order is not empty and status is "new"
    - Process each item by calling `inventory.remove_flower()`
    - Track each successful removal in processed_items
    - In except block: set status to "failed", call `_rollback_inventory()`, raise `InvalidOrderError`
    - In else block: set status to "processed", return order details dictionary
    - In finally block: log processing attempt with final status
    - Return format: `{'customer': str, 'items': dict, 'total': float, 'status': str}`
21. **Write a Python function to rollback inventory changes.**

    Define: `_rollback_inventory(self, processed_items)`

The method should:

- Use try-except-finally pattern
- Track `rollback_success` flag
- Iterate through processed_items list of (flower_name, quantity) tuples
- Add quantities back to inventory flowers
- In except block: log errors but don't raise (this is cleanup code)
- Set rollback_success to False on any errors
- In finally block: print rollback status and return success flag

22. **Write a Python function to provide string representation of order.**

    Define: `__str__(self)` The method should:

    - Create items string by joining formatted item descriptions
    - Format each item as: `"{qty} {name}"`
    - Join multiple items with ", "
    - Return format: `"Order for {customer_name}: {items_str}. Total: ${total_price:.2f} ({status})"`
    - Example: `"Order for John Smith: 5 Rose, 3 Tulip. Total: $35.42 (processed)"`

5.5 Utility Functions

23. **Write a Python function to generate daily inventory and sales report.**

    Define: `generate_daily_report(inventory)` The function should:

    - Use try-except-else-finally pattern
    - Initialize `report_data` variable to None
    - Create report dictionary with fields: `date`, `inventory_count`, `transactions`, `stock_alerts`
    - Set date as current date in YYYY-MM-DD format using `strftime("%Y-%m-%d")`
    - Count total flowers in inventory for `inventory_count`
    - Process transaction log to calculate sales (remove transactions) and restocks (add transactions)
    - Create stock alerts list for flowers with quantity < 5
    - Each alert should include: `flower`, `current_stock`, `price`
    - In except block: raise `FlowerShopException` with error code "R001"
    - In else block: set `report_data` to completed report, print success message
    - In finally block: print completion message
    - Return the complete report dictionary

24. **Write a Python function to demonstrate exception handling patterns.**

    Define: `main()` The function should:

    - Print system header and section dividers
    - Initialize empty inventory
    - Demonstrate flower creation and validation using try-except blocks
    - Show successful operations: adding flowers, creating orders, processing orders
    - Demonstrate error handling: invalid flower data, out of stock errors, expired flowers
    - Use specific exception types in except blocks: `InvalidFlowerDataError`, `OutOfStockError`, `InvalidOrderError`
    - In main try block: perform successful operations
    - In except block: handle unexpected exceptions
    - In else block: print success message and final inventory count
    - In finally block: print session end message with timestamp
    - Show both successful operations and error scenarios

6. EXECUTION STEPS TO FOLLOW:

1. Create exception hierarchy with all required exception classes

2. Implement Flower class with validation using try-except-else

3. Develop Inventory class with full try-except-else-finally blocks

4. Build Order class with transaction handling and rollback

5. Implement utility functions with complete error handling

6. Create main function demonstrating all exception handling patterns

Execution Steps to Follow:

- All actions like build, compile, running application, running test cases will be through Command Terminal.

- To open the command terminal the test takers, need to go to Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal

- This editor Auto Saves the code

● If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page)

● These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.

● To launch application: python3 filename.py

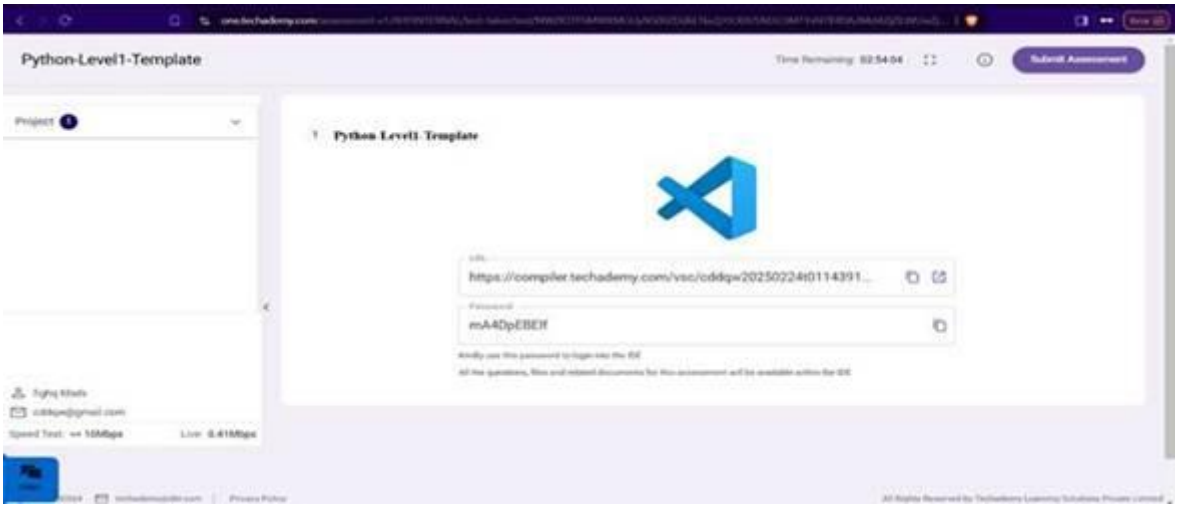
● To run Test cases: python3 -m unittest

Screen shot to run the program

To run the application

python3 filename.py

To run the testcase python3 -m unittest



● Once you are done with development and ready with submission, you may navigate to the previous tab and submit the workspace. It is mandatory to click on “Submit Assessment” after you are done with code.