Functional Requirement Specification

Stakeholders (5 points)
Identify people who are interested in the project, such as users, managers, sponsors, etc.

- Our stakeholders are our managers, employees, our suppliers, buyers, and the shipping companies that handle our shipments

Actors and Goals (5 points)
Identify the roles of people or devices that will interact directly with the system and specify their types and goals.

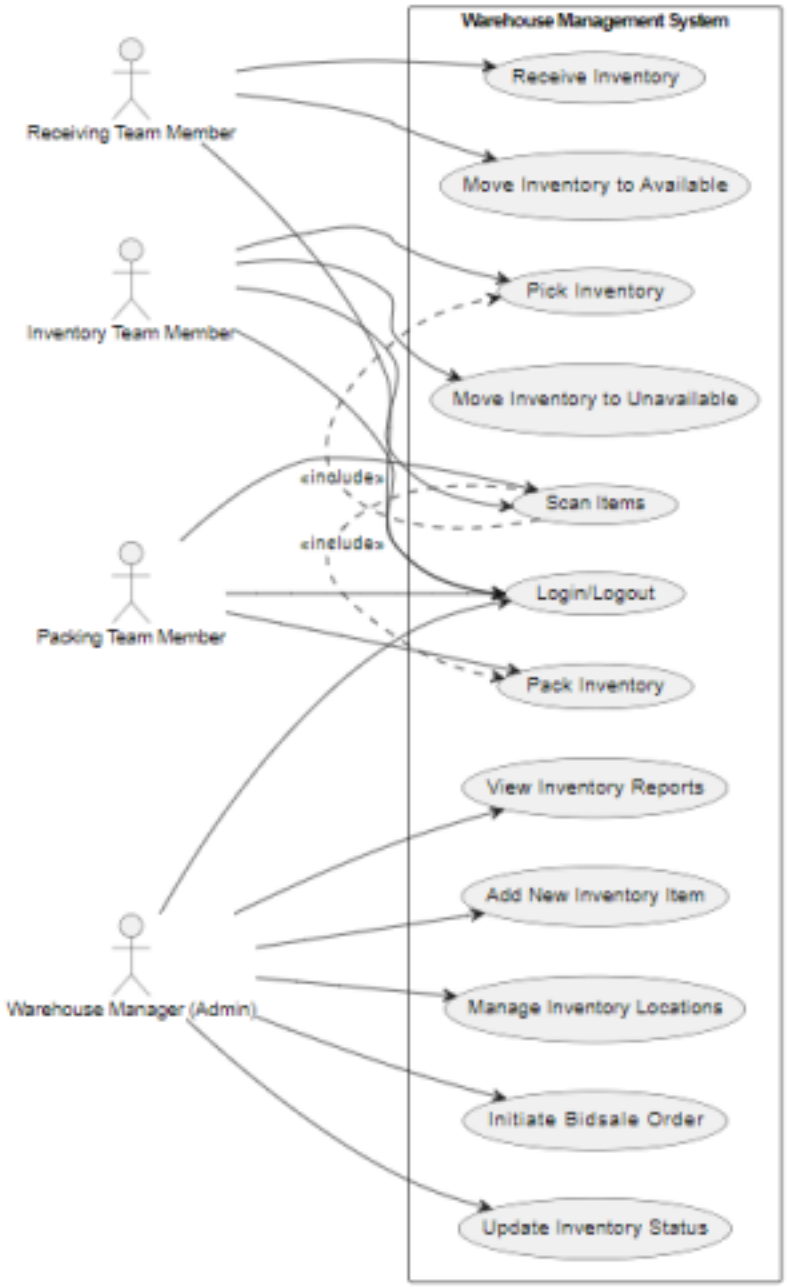
- Primary Actors
    - Receiving team members: This actor can receive items and put them away into active inventory
    - Inventory team members: This actor will get periodic orders sent and can pick items, taking them from active inventory, this action will put the items they pick into inactive inventory
    - Packing team members: This actor will scan picked items into cartons. The background process will assign this carton number to the item
    - Warehouse Manager (Admin): This actor will be able to create new users and assign/change roles to existing team members

Use Cases (10 points)
- Warehouse Manager (total: 14)
    - Add inventory location (2): To create or update a storage location within the warehouse
    - Add inventory item (2): To add a new inventory item into the system when it is received
    - Generate Bidsale order (2): To create a Bidsale order when the inventory level reaches a specific threshold
    - Move inventory status (2): To change the status of inventory items (e.g., from "Available" to "Unavailable")
    - Update item details (2): To modify the information associated with inventory items, such as quantity or location
    - Login/Logout (2): To authenticate into or out of the system
    - View inventory reports (2): To review reports on inventory status, stock levels, and order fulfillment
- Warehouse Worker (total: 12)
    - Receive inventory (2): To enter or scan the inventory items when they are received and place them in their assigned storage locations
    - Pick inventory (2): To retrieve items from their storage locations to fulfill orders
    - Pack inventory (2): To pack items into cartons for shipment, associating items

with their respective cartons
○ Scan inventory (2): To scan items for confirmation during picking or packing ○ Login/Logout (2): To access the system and record tasks under their employee ID ○ Move inventory to shipping (2): To transfer packed inventory to the "shipping" area for outbound orders

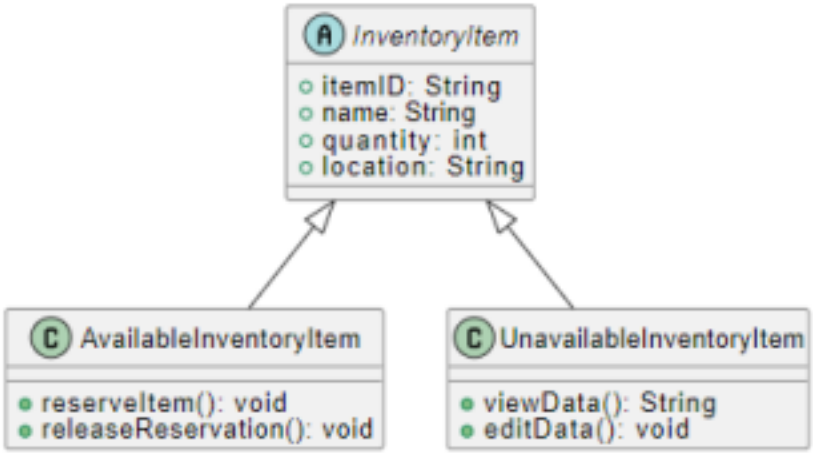
- System (total: 9)
  - ○ Guide picking process (2): To direct the warehouse worker to the correct inventory location and provide quantity information
  - ○ Confirm inventory changes (2): To validate inventory updates, such as adding, picking, or packing items
  - ○ Trigger Bidsale order (2): To automatically generate a Bidsale order when inventory reaches the defined threshold
  - ○ Archive shipped orders (3): To move completed orders to a historical data table for future reference

Use Case Diagram (10 points)

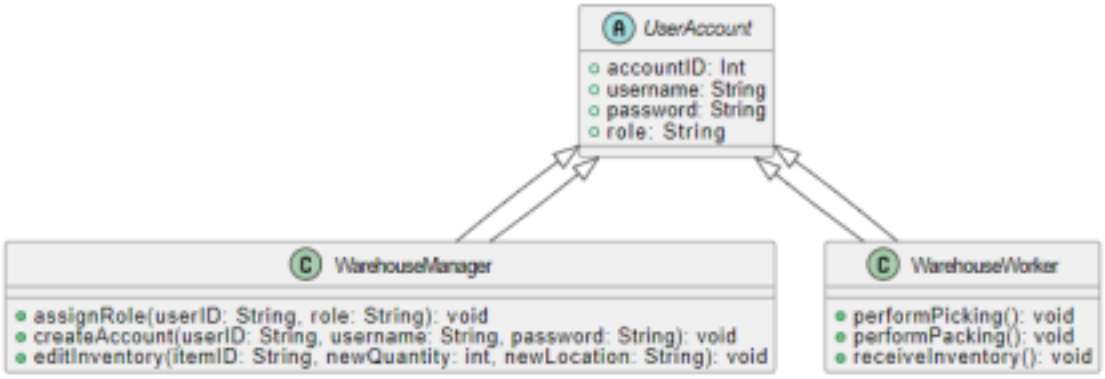


Class Diagram (20 points)

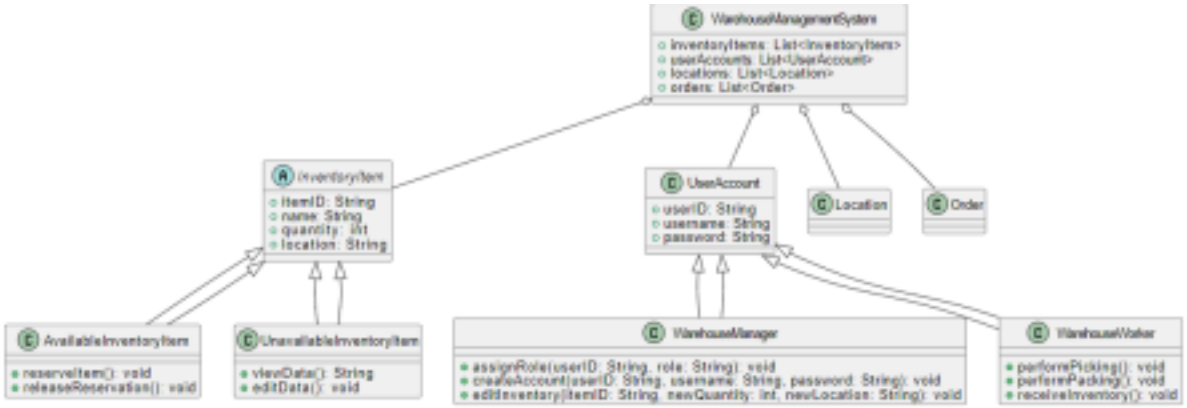
- Inventory: We have 2 types of inventory, available and unavailable. When an order is placed we will reserve the item from available so we have methods to do this. Unavailable inventory cannot be ordered, so our methods will only allow for viewing and editing



- User accounts will be an abstract where managers and workers inherit. Managers have the additional methods to edit inventory, create accounts, and remove accounts (when users are terminated). Workers will use picking, packing, and receiving functions



- As for an overview of how our system interacts, this diagram depicts how each part of the system is inherited and passes down attributes and methods. The WMS will have methods to list all items in inventory, list the user accounts, list our locations, and list orders. The remaining classes interact with the system but do not inherit the methods.



- As of this time, I do not plan to use any enumeration in my WMS. This may change as

our requirements and priorities are shifted throughout development.

Traceability Matrix

Number	Priority Weight (1-5: 1: lowest, 5: highest)	Description
REQ1	5	Administrators and workers can log in to the system
REQ2	5	Items are added to available inventory when received
REQ3	5	Items are subtracted from available inventory when picked
REQ4	4	Items are able to be packed and associated with a carton number
REQ5	3	Administrators can add or remove users
REQ6	5	Administrators can generate orders
REQ7	4	When an outbound order is available, workers are directed to locations to pick items
REQ8	2	Items that have been packed are added to a new table in the database for traceability purposes
REQ9	2	Administrators should have the ability to update inventory locations and quantities to fix discrepancies

System Architecture and Design

Architectural Styles

The WMS follows a Client-server architectural style:

- The Flask based GUI acts as the client, handling user interaction and sending requests to the backend
- The backend, using Python and MongoDB, acts as the server, processing data, executing CRUD operations, and interacting with the database

Additionally, within the backend, our system is component-based:

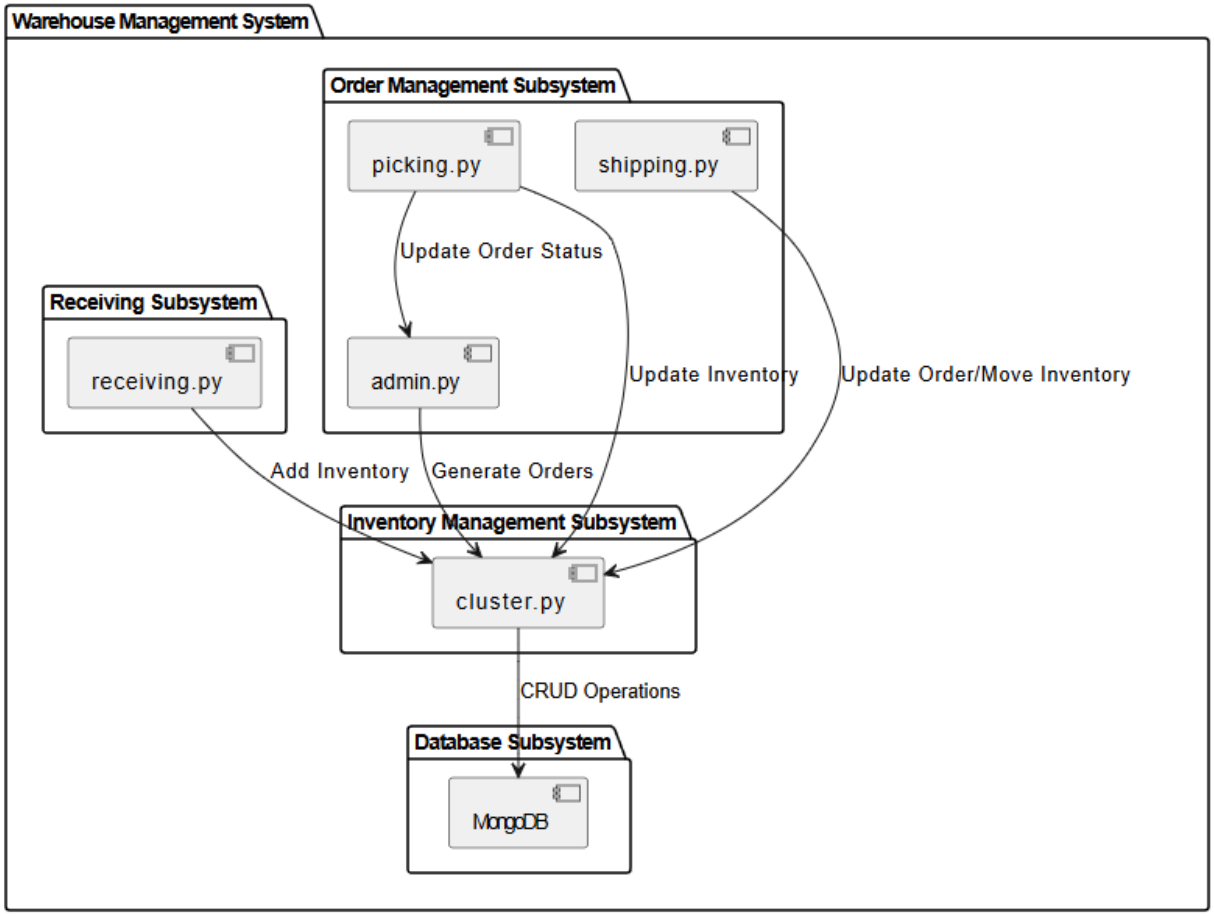
- Components like receiving.py, picking.py, and shipping.py focus on distinct functionalities, ensuring modularity and separation of concerns

Identifying Subsystems

The subsystems for the WMS are divided as follows:

- Inventory Management Subsystem: Handles operations related to active and inactive inventory (cluster.py)
- Order Management Subsystem: Processes orders, including generating, picking, and shipping (admin.py, picking.py, shipping.py)
- Receiving Subsystem: Handles adding new inventory (receiving.py)
- Database Subsystem: Manages persistent data storage using MongoDB

UML Diagram:



Mapping Subsystems to Hardware

This system will run on multiple machines:

- Client Machine: Hosts the Flask-based GUI
- Server Machine: Hosts the backend application and MongoDB database

Mapping:

- The client sends HTTP requests to the server using the Flask application
- The server processes the requests and interacts with MongoDB for data storage

Persistent Data Storage

This system *does* require persistent data storage

Persistent objects:

- Inventory items
- Orders
- Inactive inventory

Storage Management Strategy

- A NoSQL database (MongoDB) is used for its flexibility and scalability, adapting to the dynamic needs of warehousing

Network Protocol

The WMS uses HTTP for communication between the client (Flask) and the server. I chose HTTP due to it being a widely supported, simple, and reliable protocol for web-based client-server communication

Global Control Flow

Execution Order

- The system is event-driven. Users interact with specific modules (ex. receiving, picking, packing) and each action generates events to trigger appropriate backend operations

Time Dependency

- The system is event-response type with no strict real-time requirements. Timely execution of events depends on user input and interaction

Concurrency

- Currently, the system does not use multiple threads. If a scenario arises where multiple picking associates need to work on the same order simultaneously, implementing multithreading or other synchronization techniques may be necessary to avoid conflicts and ensure data consistency.

  At present, each user interacts with the system independently, with their tasks handled as separate requests by the server. The system relies on the database's built-in concurrency control to manage simultaneous operations on different orders.

Hardware Requirements

Client Requirements:

- A device capable of running a web browser (for Flask GUI)
- Minimum resolution: 1024 × 768 pixels
- Network connection to communicate with the server

Server Requirements:

- A machine capable of running Python and MongoDB.
- Minimum specs:
  - CPU: Dual-core processor
  - RAM: 4 GB
  - Storage: 20 GB (to support MongoDB and application files)
  - Network: Minimum bandwidth of 1 Mbps

User Interface Design and Implementation

Introduction

The user interface (UI) design and implementation are crucial for creating an intuitive and user-friendly application. This document outlines the design principles, components, and implementation steps for the UI of the Warehouse Management System (WMS).

Design Principles

1. **Simplicity: Keep the interface simple and uncluttered. Each screen should focus on a single task to avoid overwhelming the user.**
2. **Consistency: Maintain a consistent look and feel across the application. Use the same colors, fonts, and layout structure.**
3. **Feedback: Provide immediate feedback for user actions. Use flash messages to indicate success or failure.**
4. **Accessibility: Ensure the UI is accessible to all users, including those with disabilities. Use semantic HTML and provide alternative text for images.**
5. **Responsiveness: Design the interface to be responsive, so it works well on different screen sizes and devices.**

Components

1. **Header: Contains the application title and a navigation bar for quick access to different sections.**
2. **Main Content Area: Displays the main content for each page, such as forms and tables.**
3. **Footer: Contains copyright information and any additional links or disclaimers.**

Implementation Steps

1. **Define the Layout: Use a consistent layout structure for all pages. For example, a header at the top, a main content area in the middle, and a footer at the bottom.**
2. **Create Templates: Use HTML templates to define the structure of each page. Use Flask's render_template function to render these templates.**
3. **Style the Templates: Use CSS to style the templates. Ensure that the styles are consistent across the application.**
4. **Add Functionality: Use Flask routes to handle user actions and update the UI accordingly. Use flash messages to provide feedback to the user.**
5. **Test the UI: Test the UI on different devices and screen sizes to ensure it is responsive and user-friendly.**

Example Templates

Base Template (base.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Default Title{% endblock %}</title>
    <link rel="stylesheet" href="/static/styles.css">
</head>
<body>
    <header>
        <h1>Warehouse Management System</h1>
        {% block header %}{% endblock %}
    </header>
    <main>
        {% block content %}{% endblock %}
    </main>
    <footer>
        <p>&copy; 2024 Jonathan Fulkerson</p>
    </footer>
</body>
</html>
```

Login Template (login.html)

```
{% extends 'base.html' %}

{% block title %}Login{% endblock %}

{% block content %}
<div class="container">
    <h2>Login</h2>

    {% with messages = get_flashed_messages(with_categories=true) %}
        {% if messages %}
            <ul>
                {% for category, message in messages %}
                    <li class="{{ category }}">{{ message }}</li>
                {% endfor %}
            </ul>
        {% endif %}
    {% endwith %}
```

```
    <form action="{{ url_for('auth.login') }}" method="post">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required>

        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required>
        <div>
            <button type="submit">Login</button>
        </div>
    </form>
</div>
{% endblock %}
```

Design of Tests

Introduction

Testing is essential to ensure the quality and reliability of the application. This document outlines the design of unit tests and integration tests for the WMS.

Unit Tests

Purpose

Unit tests focus on testing individual components of the application in isolation. They ensure that each function or module works correctly.

Tools

- **pytest**: A testing framework for writing and running tests.
- **pymongo**: For interacting with the MongoDB database during tests.

Example Unit Tests

test_auth.py

```
import pytest
import cluster

def test_authenticate_user_success():
    username = "test_user"
    password = "correct_password"
    user = cluster.authenticate_user(username, password)
    assert user is not None
    assert user['username'] == username

def test_authenticate_user_failure():
    username = "test_user"
```

```
    password = "wrong_password"
    user = cluster.authenticate_user(username, password)
    assert user is None

test_inventory.py
import pytest
import cluster

def test_add_inventory_item():
    upc = "123456789012"
    quantity = 10
    location = "Aisle 1"
    reserved = False
    result = cluster.add_inventory_item(upc, quantity, location, reserved)
    assert result is not None

def test_find_item():
    upc = "123456789012"
    item = cluster.find_item(upc)
    assert item is not None
    assert item['upc'] == upc
```

Integration Tests

Purpose
Integration tests focus on testing the interactions between different components of the application. They ensure that the components work together as expected.

Tools

- **pytest**: A testing framework for writing and running tests.
- **requests**: For making HTTP requests to the application during tests.

Example Integration Tests
test_endpoints.py

```
import pytest
from app import app

@pytest.fixture
def client():
    with app.test_client() as client:
        yield client
```

```
def test_login_endpoint(client):
    response = client.post('/auth/login', data=dict(username='test_user',
password='correct_password'))
    assert response.status_code == 302  # Redirects after successful login

def test_login_endpoint_failure(client):
    response = client.post('/auth/login', data=dict(username='test_user',
password='wrong_password'))
    assert response.status_code == 302  # Redirects after failed login
```

test_workflow.py

```
import pytest
from app import app

@pytest.fixture
def client():
    with app.test_client() as client:
        yield client

def test_picking_workflow(client):
    # Login first
    client.post('/auth/login', data=dict(username='test_user', password='correct_password'))

    # Begin picking
    response = client.post('/picking/', data=dict(action='begin_picking'))
    assert response.status_code == 200
    assert b'Confirm Picking' in response.data

    # Confirm picking
    response = client.post('/picking/confirm', data=dict(order_id='00000001',
upc_123456789012=1))
    assert response.status_code == 302  # Redirects after successful confirmation
    assert b'Order 00000001 successfully picked.' in response.data
```

Conclusion

By following the above guidelines for UI design and implementation, along with the design of unit and integration tests, you can ensure that your Warehouse Management System is both user-friendly and reliable.