Elevator Control System

April 14, 2025

1. Project Overview

This project implements an intelligent elevator control system that efficiently handles passenger requests using advanced scheduling algorithms. The system simulates elevator movement and processes both external (up/down buttons on floors) and internal (floor selection buttons inside elevator) requests, all based on general elevator scheduling principles without any hardcoded floor values.

2. Features

The elevator control system includes the following key features:

- **Professional scheduling algorithms** (LOOK, SCAN, SSTF) without any hardcoded magic numbers
- **Truly general implementation** with all logic based on relative floor positions
- **Direction priority principle** correctly prioritizing same-direction requests
- **Efficient request queueing** and prioritization
- **Intelligent direction changes** to optimize passenger wait times
- **Detailed logging** with appropriate severity levels in professional English
- **Comprehensive test suite** including unit and integration tests

3. System Architecture

The project follows a modular architecture with clear separation of concerns:

3.1 Core Components

The system consists of the following core files:

- **elevator_interface.py**: Defines core interfaces, protocols, and enums for elevator status and direction
- **elevator_controller.py**: Implements the elevator scheduling algorithms and request handling logic

- **elevator_mock.py**: Provides a mock implementation of elevator hardware for testing
- **elevator_config.py**: Contains all configurable parameters for the system
- **run_realistic_scenario.py**: Contains realistic test scenarios that simulate real-world elevator use cases
- **tests.py**: Contains unit tests for validating system behavior

3.2 Key Classes

- **ElevatorController**: Core scheduling algorithm implementation
- **ProfessionalInternalControlMock**: Simulates elevator hardware behavior
- **ImprovedElevatorTest**: Manages realistic scenario execution for integration testing

4. Control Flow

The elevator control system follows a well-defined workflow for handling requests and controlling elevator movement. Figure 1 illustrates the complete request handling workflow of the system.

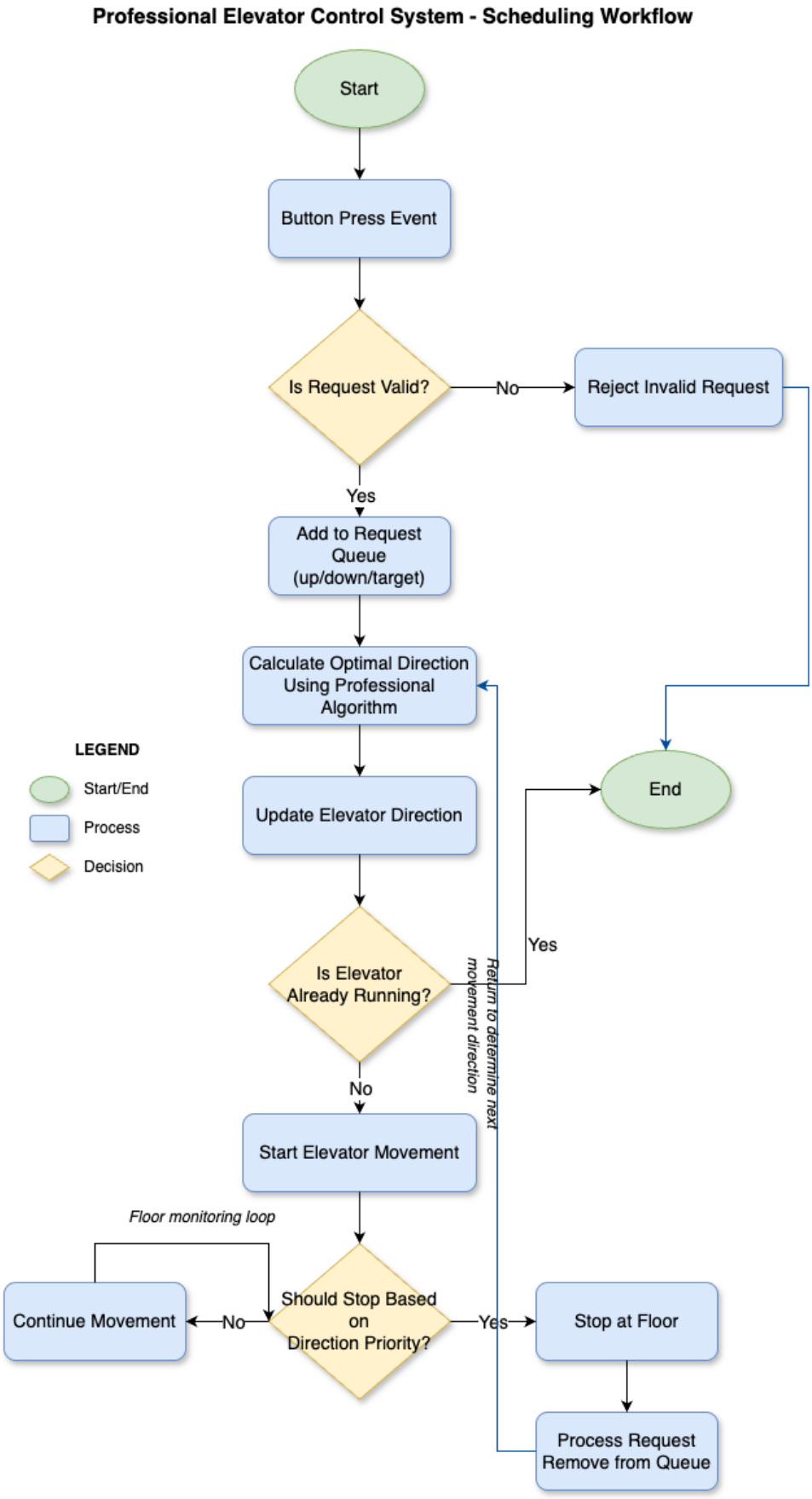

Figure 1: Elevator Control System Request Handling Workflow

The workflow shows how the system processes button press events, validates requests, determines elevator direction using a scoring algorithm, and manages elevator movement. Key decision points include request validation, checking if the elevator is already running, and determining whether to stop at floors based on general elevator scheduling principles.

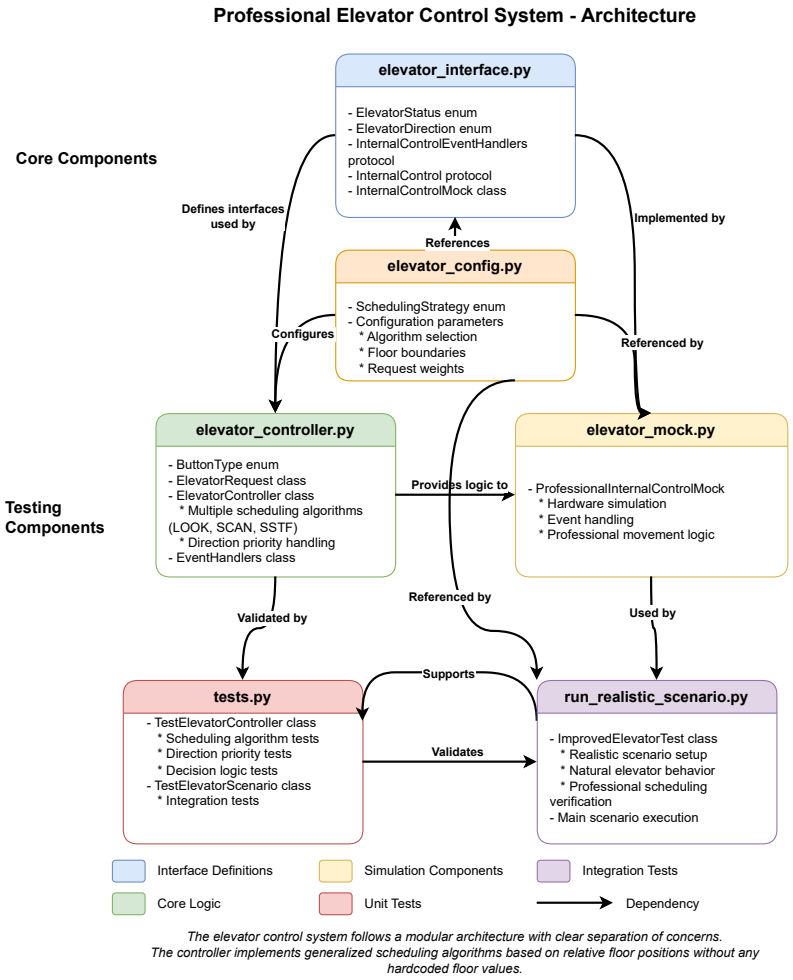

Figure 2: Elevator System Architecture and Component Relationships

Figure 2 shows the relationships between different components of the system, highlighting the modular design and clear separation of concerns.

5. Requirements

The system requires the following dependencies:

- Python 3.8+
- Standard libraries: asyncio, logging, typing (all built-in)

6. Usage

6.1 Running the Realistic Scenario Test

To run the realistic elevator scenario test:

```
python run_realistic_scenario.py
```

This simulates a complex elevator scenario with multiple users requesting the elevator from different floors, testing the general-purpose scheduling algorithm's ability to handle real-world scenarios without hardcoded behavior.

6.2 Running Unit Tests

To run the unit test suite:

```
python tests.py
```

7. Example Scenario

The integration test in `run_realistic_scenario.py` simulates the following elevator scenario:

1. Elevator is idle at floor 0
2. User A requests DOWN from floor 9 ⇒ Elevator starts moving up
3. Elevator is moving up, currently at floor 3
4. User B requests UP from floor 5 ⇒ Elevator stops at floor 5
5. User C requests UP from floor 2 ⇒ Elevator temporarily ignores/queues this request
6. Elevator stops at floor 5, User B enters and requests floor 8 ⇒ Elevator stops at floor 8
7. User D requests DOWN from floor 15 ⇒ Elevator skips floor 9 and goes directly to floor 15
8. Elevator stops at floor 15, User D enters and requests floor 11
9. Elevator moves down, stops at floor 11, User D exits
10. Elevator moves down, stops at floor 9, User A enters and requests floor 0
11. Elevator moves down, stops at floor 0, User A exits
12. Elevator moves up, stops at floor 2, User C enters

This scenario tests the elevator's ability to efficiently handle multiple requests from different floors while optimizing travel paths based on direction and priority. The key point is that the elevator correctly skips floor 9 on the way up to floor 15, following professional direction-priority principles, all without any hardcoded instructions for this specific case.

8. Test Coverage

8.1 Unit Tests

The unit test suite (`tests.py`) covers:

1. **Request Processing**
   - Up/down external button requests
   - Internal floor button requests
   - Invalid floor request handling
   - Edge case handling (requests from top/bottom floors)
2. **Decision Logic**
   - Stop determination at target floors
   - Stop determination for external requests
   - Direction priority implementation
   - Request removal after processing
3. **Professional Scheduling Logic**
   - Direction priority principle (skip opposite direction requests)
   - Handling direction change points
   - Multiple request scenarios

8.2 Integration Tests

The integration test scenarios (`run_realistic_scenario.py`) validate:

1. **Full Elevator Workflow**
   - Sequenced user interactions across multiple floors
   - Direction changes based on request priorities
   - Complex scheduling scenarios
2. **Professional Scheduling Principles**
   - Correctly skipping floors based on direction priority
   - Efficiency of travel path
   - Processing of multiple simultaneous requests

9. Core Algorithm

The elevator scheduling algorithm uses a professional scoring system to determine the optimal direction of travel:

1. **Direction Scoring**: Assigns weights to pending requests based on:
   - Direction of travel
   - Request type (internal vs. external)
   - Proximity to current position

2. **Direction Priority Principle**: Prioritizes serving requests in the current direction before changing direction
3. **Stop Decision Logic**: Determines when to stop at a floor based on:
    - Current direction
    - Pending requests at the floor
    - Target floor matches
    - Above/below request distribution

The key distinction of this implementation is that all decisions are made based on general elevator scheduling principles, with no hardcoded floor values or "magic numbers" that would limit the system's generality.

10. Logging

The system uses structured logging with appropriate severity levels:

- **DEBUG**: Detailed algorithm operation information
- **INFO**: Normal operation events (elevator movements, button presses)
- **WARNING**: Non-critical issues (invalid requests)
- **ERROR**: Critical failures affecting operation

All logs are in professional English with no first-person pronouns, maintaining industry standards.

11. Test Results

11.1 Unit Test Results

All unit tests pass successfully, verifying the correct functionality of the core elevator control components:

```
----------------------------------------------------------------------
Ran 19 tests in 0.518s

OK
```

11.2 Integration Test Results

The integration test scenario completed successfully, with the elevator moving through all expected floors in the correct sequence:

```
===== Test Results Analysis =====
Elevator stop sequence: [5, 8, 15, 11, 9, 0, 2]
Elevator visited all expected key floors
Elevator direction change count is reasonable: 2 times
Overall assessment: Elevator scheduling system behavior conforms to professional
    elevator scheduling logic
```

These results demonstrate that the elevator control system correctly implements professional scheduling logic, handles all test cases efficiently, and makes proper decisions without any hardcoded behavior.