Elevator Simulation - Low Level Design

1. File Structure

```
elevator/
├── CMakeLists.txt
├── docs/
│   ├── HLD.md
│   └── LLD.md
├── include/
│   ├── Types.hpp          # Enums, constants, aliases
│   ├── Domain.hpp         # Elevator, Floor, Building
│   ├── EventQueue.hpp     # Thread-safe queue
│   ├── Scheduler.hpp      # IScheduler + both controllers
│   └── Simulation.hpp     # Engine + Logger
├── src/
│   ├── main.cpp
│   ├── Domain.cpp
│   ├── Scheduler.cpp
│   └── Simulation.cpp
└── tests/
    ├── UnitTests.cpp
    └── StressTests.cpp
```

---

2. Types.hpp - Core Definitions

Enums

```cpp
enum class Direction { Up, Down, Idle };

enum class ElevatorState {
    Idle,           // Stationary, no pending requests
    Moving,         // Traveling between floors
    DoorsOpening,   // Arrived, opening doors
    DoorsOpen,      // Passengers boarding/alighting
    DoorsClosing    // Preparing to move
};

enum class EventType {
    HallCall,       // Floor button pressed
    CarCall,        // Destination selected in car
    ElevatorArrived,// Elevator reached a floor
    DoorsOpened,
    DoorsClosed,
    Tick,           // Simulation time advance
    Shutdown        // Graceful termination
};
```

```
enum class ControllerType { Master, Distributed };
```

Configuration Struct

```
struct Config {
    int numFloors = 10;
    int numElevators = 3;
    int carCapacity = 6;
    int tickDurationMs = 500;
    int doorOpenTicks = 3;
    int floorTravelTicks = 2;
    ControllerType controllerType = ControllerType::Master;
};
```

Event Struct

```
struct Event {
    EventType type;
    int floor = -1;
    int elevatorId = -1;
    Direction direction = Direction::Idle;
    std::chrono::steady_clock::time_point timestamp;
};
```

---

3. Domain.hpp - Core Domain Classes

3.1 Elevator Class

```
class Elevator {
private:
    int id_;
    int currentFloor_;
    Direction direction_;
    ElevatorState state_;
    std::set<int> carCalls_;        // Destinations (sorted, no dups)
    int passengerCount_;
    int capacity_;

    mutable std::mutex mutex_;       // Protects state

public:
    // Constructor
    Elevator(int id, int capacity, int startFloor = 1);
```

```
    // Getters (thread-safe)
    int getId() const;
    int getCurrentFloor() const;
    Direction getDirection() const;
    ElevatorState getState() const;
    int getPassengerCount() const;

    // Commands
    void addCarCall(int floor);
    void removeCarCall(int floor);
    bool hasCarCallAt(int floor) const;

    // State transitions
    void startMoving(Direction dir);
    void arriveAtFloor(int floor);
    void openDoors();
    void closeDoors();
    void setIdle();

    // Query helpers
    bool hasCallsAbove() const;
    bool hasCallsBelow() const;
    std::optional<int> getNextStop() const;
    int costToServe(int floor, Direction dir) const;
};
```

3.2 Floor Class

```
class Floor {
private:
    int floorNumber_;
    bool upButtonPressed_;
    bool downButtonPressed_;

public:
    Floor(int number);

    void pressUpButton();
    void pressDownButton();
    void clearUpButton();
    void clearDownButton();

    bool isUpPressed() const;
    bool isDownPressed() const;
    int getNumber() const;
};
```

3.3 Building Class

```
class Building {
private:
    std::vector<Floor> floors_;
    std::vector<std::unique_ptr<Elevator>> elevators_;
    Config config_;
    mutable std::mutex mutex_;

public:
    Building(const Config& config);

    // Accessors
    int getNumFloors() const;
    int getNumElevators() const;
    Elevator& getElevator(int id);
    const Elevator& getElevator(int id) const;
    Floor& getFloor(int number);

    // Hall call management
    void registerHallCall(int floor, Direction dir);
    void clearHallCall(int floor, Direction dir);
    bool hasHallCall(int floor, Direction dir) const;
    std::vector<std::pair<int, Direction>> getAllHallCalls() const;
};
```

---

4. EventQueue.hpp - Thread-Safe Queue

```
template<typename T>
class EventQueue {
private:
    std::queue<T> queue_;
    mutable std::mutex mutex_;
    std::condition_variable cv_;
    std::atomic<bool> shutdown_{false};

public:
    // Add event to queue
    void push(T event) {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            queue_.push(std::move(event));
        }
        cv_.notify_one();
    }

    // Wait and retrieve event
    std::optional<T> pop() {
        std::unique_lock<std::mutex> lock(mutex_);
        cv_.wait(lock, [this] {
            return !queue_.empty() || shutdown_.load();
```

```
        });

        if (shutdown_ && queue_.empty()) {
            return std::nullopt;
        }

        T event = std::move(queue_.front());
        queue_.pop();
        return event;
    }

    // Non-blocking try
    std::optional<T> tryPop() {
        std::lock_guard<std::mutex> lock(mutex_);
        if (queue_.empty()) return std::nullopt;
        T event = std::move(queue_.front());
        queue_.pop();
        return event;
    }

    void shutdown() {
        shutdown_ = true;
        cv_.notify_all();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(mutex_);
        return queue_.empty();
    }
};
```

---

5. Scheduler.hpp - Controller Classes

5.1 Interface

```
class IScheduler {
public:
    virtual ~IScheduler() = default;

    virtual void handleHallCall(int floor, Direction dir) = 0;
    virtual void handleCarCall(int elevatorId, int floor) = 0;
    virtual void onElevatorArrived(int elevatorId, int floor) = 0;
    virtual void onDoorsOpened(int elevatorId, int floor) = 0;
    virtual void onDoorsClosed(int elevatorId) = 0;
    virtual void tick() = 0;  // Called each simulation tick
};
```

5.2 Master Controller

```
class MasterController : public IScheduler {
private:
    Building& building_;
    EventQueue<Event>& eventQueue_;

    // Assignment tracking: floor+direction -> assigned elevator
    std::map<std::pair<int, Direction>, int> assignments_;
    mutable std::mutex mutex_;

public:
    MasterController(Building& building, EventQueue<Event>& queue);

    void handleHallCall(int floor, Direction dir) override;
    void handleCarCall(int elevatorId, int floor) override;
    void onElevatorArrived(int elevatorId, int floor) override;
    void onDoorsOpened(int elevatorId, int floor) override;
    void onDoorsClosed(int elevatorId) override;
    void tick() override;

private:
    // Find best elevator for a hall call
    int selectElevator(int floor, Direction dir);

    // Calculate cost for elevator to serve request
    int calculateCost(const Elevator& elev, int floor, Direction dir);

    // Determine next action for elevator
    void dispatchElevator(int elevatorId);
};
```

Master Controller Algorithm (LOOK):

1. On hall call: Find elevator with lowest cost, assign exclusively
2. Cost formula: `|currentFloor - targetFloor| + penalty_if_wrong_direction`
3. Elevator continues in direction until no more calls ahead
4. Then reverses if calls in opposite direction, else goes idle

5.3 Distributed Controller

```
class DistributedController : public IScheduler {
private:
    Building& building_;
    EventQueue<Event>& eventQueue_;

    // Claim board: floor+direction -> claiming elevator (-1 if unclaimed)
    std::map<std::pair<int, Direction>, int> claimBoard_;
    mutable std::mutex mutex_;

public:
    DistributedController(Building& building, EventQueue<Event>& queue);
```

```
    void handleHallCall(int floor, Direction dir) override;
    void handleCarCall(int elevatorId, int floor) override;
    void onElevatorArrived(int elevatorId, int floor) override;
    void onDoorsOpened(int elevatorId, int floor) override;
    void onDoorsClosed(int elevatorId) override;
    void tick() override;

private:
    // Each elevator tries to claim unclaimed calls
    void tryClaimCalls(int elevatorId);

    // Release claim when served
    void releaseClaim(int floor, Direction dir);

    // Check if this elevator has the claim
    bool hasClaim(int elevatorId, int floor, Direction dir);
};
```

Distributed Controller Algorithm:

1. Hall calls posted to shared claim board (initially unclaimed)
2. Each elevator checks board on tick, claims nearest unclaimed call
3. Claim is atomic (mutex protected) - first to claim wins
4. After serving, elevator releases claim
5. Prevents duplicate service via claim ownership

---

6. Simulation.hpp - Engine & Logger

6.1 Logger Class

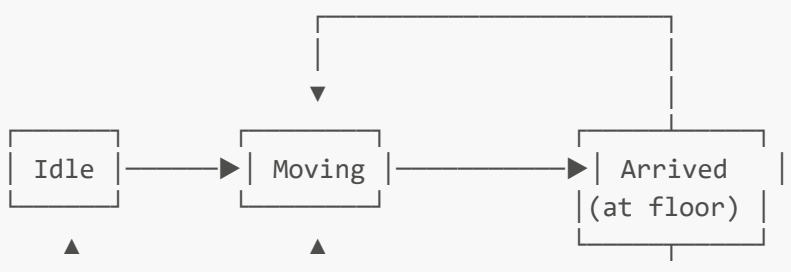
```
class Logger {
private:
    std::mutex mutex_;
    std::ostream& out_;
    bool enabled_;

public:
    Logger(std::ostream& out = std::cout, bool enabled = true);

    void log(const std::string& message);
    void logEvent(const Event& event);
    void logElevatorState(const Elevator& elev);

    void enable();
    void disable();
};
```

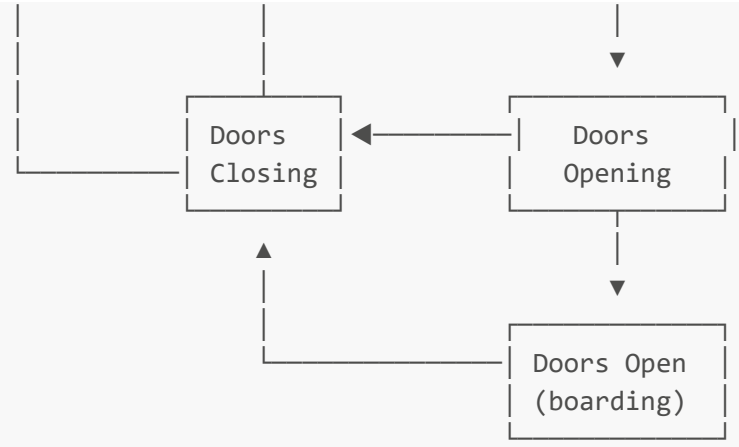
6.2 Simulation Engine

```
class SimulationEngine {
private:
    Building building_;
    std::unique_ptr<IScheduler> scheduler_;
    EventQueue<Event> eventQueue_;
    Logger logger_;
    Config config_;

    std::vector<std::thread> elevatorThreads_;
    std::atomic<bool> running_{false};
    std::atomic<int> currentTick_{0};

public:
    SimulationEngine(const Config& config);
    ~SimulationEngine();

    // Control
    void start();
    void stop();
    bool isRunning() const;

    // Commands (from CLI)
    void requestHallCall(int floor, Direction dir);
    void requestCarCall(int elevatorId, int floor);

    // Status
    void printStatus() const;
    int getCurrentTick() const;

private:
    void runSimulationLoop();
    void runElevatorLoop(int elevatorId);
    void processTick();
    void processEvent(const Event& event);

    void createScheduler();
};
```

---

7. State Machine - Elevator






```
Transitions:
- Idle → Moving: When assigned a call
- Moving → Arrived: Reached target floor
- Arrived → DoorsOpening: Always (auto-transition)
- DoorsOpening → DoorsOpen: After open delay
- DoorsOpen → DoorsClosing: After boarding time
- DoorsClosing → Moving: If more stops
- DoorsClosing → Idle: If no more stops
```

---

8. Thread Synchronization

Shared Resources & Protection

Resource	Protected By	Accessed From
Building state	`Building::mutex_`	All threads
Elevator state	`Elevator::mutex_`	Elevator thread, Controller
Event queue	`EventQueue::mutex_`	All threads
Claim board	`DistributedController::mutex_`	Controller only
Assignments	`MasterController::mutex_`	Controller only

Lock Ordering (to prevent deadlock)

1. EventQueue mutex
2. Building mutex
3. Elevator mutex (by ID order if multiple)
4. Controller mutex

---

9. Key Algorithms

9.1 Cost Calculation (Master Controller)

```
int calculateCost(const Elevator& e, int targetFloor, Direction callDir) {
    int distance = std::abs(e.getCurrentFloor() - targetFloor);

    if (e.getState() == ElevatorState::Idle) {
        return distance;  // Base cost
    }

    bool sameDirection = (e.getDirection() == callDir);
    bool onTheWay = (e.getDirection() == Direction::Up && targetFloor >
e.getCurrentFloor())
                 || (e.getDirection() == Direction::Down && targetFloor <
e.getCurrentFloor());

    if (sameDirection && onTheWay) {
        return distance;  // Best case: pick up on the way
    }

    // Penalty for wrong direction or need to reverse
    return distance + 2 * config_.numFloors;
}
```

9.2 Next Stop Selection (LOOK Algorithm)

```
std::optional<int> getNextStop(const Elevator& e, const Building& b) {
    int current = e.getCurrentFloor();
    Direction dir = e.getDirection();

    // Collect all stops: car calls + assigned hall calls
    std::set<int> stops = e.getCarCalls();
    // Add hall calls assigned to this elevator...

    if (dir == Direction::Up) {
        // Find nearest stop above
        auto it = stops.upper_bound(current);
        if (it != stops.end()) return *it;
        // None above, check below (reverse)
        if (!stops.empty()) return *stops.rbegin();
    } else if (dir == Direction::Down) {
        // Find nearest stop below
        auto it = stops.lower_bound(current);
        if (it != stops.begin()) return *std::prev(it);
        // None below, check above (reverse)
        if (!stops.empty()) return *stops.begin();
    }

    return std::nullopt;  // No stops
}
```

---

10. Testing Strategy

Unit Tests (UnitTests.cpp)

```
// Elevator Tests
TEST(ElevatorTest, InitialState)
TEST(ElevatorTest, AddCarCall)
TEST(ElevatorTest, StateTransitions)
TEST(ElevatorTest, DirectionLogic)

// Controller Tests
TEST(MasterControllerTest, AssignNearestElevator)
TEST(MasterControllerTest, NoDoubleAssignment)
TEST(MasterControllerTest, LOOKAlgorithm)

TEST(DistributedControllerTest, ClaimMechanism)
TEST(DistributedControllerTest, NoDuplicateService)

// EventQueue Tests
TEST(EventQueueTest, PushPop)
TEST(EventQueueTest, ThreadSafety)
TEST(EventQueueTest, Shutdown)
```

Stress Tests (StressTests.cpp)

```
// High volume test
TEST(StressTest, HighTraffic) {
    // 100 random requests in rapid succession
    // Verify: all served, no deadlock, no crash
}

// Concurrency test
TEST(StressTest, ConcurrentAccess) {
    // Multiple threads generating requests
    // Run with ThreadSanitizer
}

// Long running test
TEST(StressTest, Endurance) {
    // Run for 10000 ticks
    // Check memory stability
}
```

---

11. Error Handling

Scenario	Handling

Scenario	Handling
Invalid floor number	Return early, log warning
Invalid elevator ID	Return `std::nullopt` or log error
Queue shutdown during wait	Return `std::nullopt`
Capacity exceeded	Reject boarding, log event

Using `std::optional` for "might not exist" returns, avoiding exceptions for flow control.