Modeling
v1.0

Warehouse Robot Management System

Contents

1 Changelog **2**

2 System Architecture Overview **2**

3 Class Design **2**

3.1 Core System Classes . . . 2

3.1.1 Robot (Abstract Base Class) . . . 2

3.1.2 Battery Class . . . 2

3.1.3 Operator Class . . . 3

3.2 Specialized Robot Implementations . . . 3

3.2.1 CarrierRobot . . . 3

3.2.2 ScannerRobot . . . 3

3.2.3 SorterRobot . . . 4

3.3 Supporting Classes . . . 4

3.3.1 Position Class . . . 4

3.3.2 Task Class . . . 4

3.3.3 Item Class . . . 5

3.3.4 InventoryDatabase Class . . . 5

4 System Workflows **6**

4.1 Task Assignment Process . . . 6

4.2 Battery Management Process . . . 7

4.3 Robot-Specific Operations . . . 8

4.3.1 CarrierRobot Transportation . . . 8

4.3.2 ScannerRobot Inventory Scanning . . . 8

4.3.3 SorterRobot Package Organization . . . 9

5 Implementation Guidelines **11**

5.1 Development Phase 1: Core Infrastructure . . . 11

5.2 Development Phase 2: Robot Specializations . . . 11

5.3 Development Phase 3: Management Systems . . . 11

5.4 Testing Strategy . . . 11

6 Configuration and Deployment **12**

6.1 System Configuration . . . 12

6.2 Performance Considerations . . . 12

1 Changelog

- v1.0: Initial system design and modeling specifications.

2 System Architecture Overview

Our warehouse robot management system coordinates three types of autonomous robots (Carrier, Scanner, Sorter) working together under operator supervision. The system uses object-oriented design with clear inheritance hierarchies, composition relationships for critical components like batteries, and aggregation for flexible operator-robot assignments.

The architecture supports real-time operations with polymorphic behavior allowing different robot types to execute specialized tasks while maintaining a consistent interface. Battery management, position tracking, and task assignment are integrated throughout the system to ensure reliable warehouse automation.

3 Class Design

3.1 Core System Classes

3.1.1 Robot (Abstract Base Class)

The Robot class serves as the foundation for all robot types in our warehouse system. It defines the common interface and shared functionality while requiring specialized implementations for task execution.

- **Protected Attributes**:
  - `robot_id_`: Unique identifier for each robot instance.
  - `model_`: Robot model specification for maintenance and capabilities.
  - `operational_status_`: Current operational state (IDLE, ACTIVE, CHARGING, MAINTENANCE, ERROR).
- **Abstract Methods** (must be implemented by all robot types):
  - `execute_task()`: Performs robot-specific operations.
  - `get_task_description()`: Returns description of current task capabilities.
- **Concrete Methods** (inherited by all robots):
  - `move()`: Basic navigation functionality.
  - `get_status()`: Returns current operational status.
  - `assign_operator()`: Associates robot with an operator.
  - `assign_battery()`: Assigns battery to robot.
  - `update_position()`: Updates robot location.
- **Protected Methods** (for derived class use):
  - `validate_operation()`: Checks if operation is safe to perform.
  - `log_activity()`: Records robot activities for audit trail.

3.1.2 Battery Class

Each robot owns exactly one battery (composition relationship). The battery cannot exist independently of its robot and is responsible for all power management operations.

- **Private Attributes**:
  - `capacity_`: Maximum battery capacity in appropriate units.
  - `charge_level_`: Current charge level (0.0 to capacity).
  - `charging_status_`: Current charging state (CHARGING, FULL, DISCHARGING, CRITICAL).
  - `battery_id_`: Unique battery identifier for tracking.
- **Public Methods**:
  - `charge()`: Initiates charging process.
  - `discharge()`: Reduces charge level during operations.

  - `get_charge_level()`: Returns current charge level.
  - `get_remaining_time()`: Estimates operational time remaining.
  - `is_low_battery()`: Checks if charge is below safe threshold.
- **Private Methods**:
  - `update_status()`: Internal status management.

3.1.3 Operator Class

Operators manage multiple robots through an aggregation relationship. Robots can exist without operators, but operators control robot task assignments and monitoring.

- **Private Attributes**:
  - `name_`: Operator's name for identification.
  - `operator_id_`: Unique operator identifier.
  - `active_session_`: Boolean indicating if operator is currently logged in.
- **Public Methods**:
  - `login()`: Authenticates operator and starts session.
  - `logout()`: Ends operator session.
  - `assign_task()`: Creates and assigns tasks to robots.
  - `monitor_robots()`: Checks status of assigned robots.
  - `emergency_stop()`: Immediately stops all assigned robots.
- **Private Methods**:
  - `validate_credentials()`: Internal authentication validation.

3.2 Specialized Robot Implementations

3.2.1 CarrierRobot

Inherits from Robot and specializes in transporting items throughout the warehouse. Implements transportation-specific task execution and load management.

- **Private Attributes**:
  - `load_capacity_`: Maximum weight the robot can carry.
  - `current_load_`: Current weight being carried.
  - `cargo_manifest_`: List of items currently being transported.
- **Public Methods**:
  - `load_item()`: Adds item to robot cargo with weight validation.
  - `unload_item()`: Removes item from robot cargo.
  - `calculate_route()`: Computes optimal path to destination.
  - `check_weight_limits()`: Validates current load against capacity.
  - `execute_task()`: Implements transportation task execution.
  - `get_task_description()`: Returns "Transportation" task type.
- **Private Methods**:
  - `optimize_load_distribution()`: Balances cargo for stability.

3.2.2 ScannerRobot

Inherits from Robot and specializes in inventory management through barcode scanning. Implements scanning-specific task execution and database integration.

- **Private Attributes**:
  - `scanner_range_`: Maximum effective scanning distance.
  - `scan_accuracy_`: Accuracy percentage for scan validation.
  - `inventory_buffer_`: Temporary storage for scanned items.
- **Public Methods**:
  - `scan_barcode()`: Reads barcode and returns item information.

  - `update_inventory()`: Updates warehouse inventory database.
  - `validate_scan()`: Verifies scan accuracy and completeness.
  - `sync_database()`: Synchronizes local data with central database.
  - `execute_task()`: Implements scanning task execution.
  - `get_task_description()`: Returns "Inventory Scanning" task type.
- **Private Methods**:
  - `calibrate_scanner()`: Internal scanner calibration.

3.2.3 SorterRobot

Inherits from Robot and specializes in organizing packages and items by category. Implements sorting-specific task execution and zone management.

- **Private Attributes**:
  - `sort_accuracy_`: Accuracy percentage for sorting operations.
  - `processing_queue_`: Queue of items waiting to be sorted.
- **Public Methods**:
  - `categorize_item()`: Determines appropriate category for item.
  - `apply_sorting_rules()`: Applies sorting logic to item.
  - `move_to_zone()`: Transports item to designated zone.
  - `update_sort_statistics()`: Records sorting performance metrics.
  - `execute_task()`: Implements sorting task execution.
  - `get_task_description()`: Returns "Package Sorting" task type.
- **Private Methods**:
  - `analyze_item_properties()`: Internal item analysis.

3.3 Supporting Classes

3.3.1 Position Class

Robots own their position (composition relationship). Position tracks location and provides spatial calculation methods.

- **Private Attributes**:
  - `x_`, `y_`: Coordinate position in warehouse.
  - `zone_`: Warehouse zone identifier.
  - `timestamp_`: Time of last position update.
- **Public Methods**:
  - `calculate_distance()`: Computes distance to another position.
  - `is_valid_position()`: Validates position within warehouse bounds.
  - `update_coordinates()`: Updates position coordinates.

3.3.2 Task Class

Tasks have an aggregation relationship with robots - they can be reassigned between robots as needed.

- **Private Attributes**:
  - `task_id_`: Unique task identifier.
  - `task_type_`: Type of task (TRANSPORT, SCAN, SORT, MAINTENANCE).
  - `priority_`: Task priority (LOW, NORMAL, HIGH, URGENT).
  - `status_`: Current task status (CREATED, ASSIGNED, IN_PROGRESS, COMPLETED, FAILED).
- **Public Methods**:
  - `assign_to_robot()`: Assigns task to specific robot.
  - `update_status()`: Changes task status.
  - `calculate_completion_time()`: Estimates completion time.

3.3.3 Item Class

Represents warehouse inventory items that robots interact with during operations.

- **Private Attributes**:
  - `item_id_`: Unique item identifier.
  - `weight_`: Physical weight of item.
  - `category_`: Item classification.
  - `location_`: Current warehouse location.
- **Public Methods**:
  - `get_weight()`: Returns item weight.
  - `get_category()`: Returns item category.
  - `set_location()`: Updates item location.

3.3.4 InventoryDatabase Class

Manages persistent storage and synchronization of warehouse inventory data. The ScannerRobot uses this class to update inventory records after scanning operations.

- **Private Attributes**:
  - `database_connection_`: Connection string for database access.
  - `inventory_records_`: In-memory cache of inventory items for quick access.
  - `last_sync_time_`: Timestamp of last successful synchronization.
- **Public Methods**:
  - `sync_database()`: Synchronizes inventory updates from robots.
  - `validate_inventory_changes()`: Validates data integrity before updates.
  - `update_inventory_records()`: Updates stored inventory records.
  - `get_item_info()`: Retrieves item information by identifier.
  - `process_inventory_changes()`: Processes pending inventory modifications.
- **Private Methods**:
  - `establish_connection()`: Internal database connection management.
  - `log_database_operation()`: Audit trail logging for database operations.

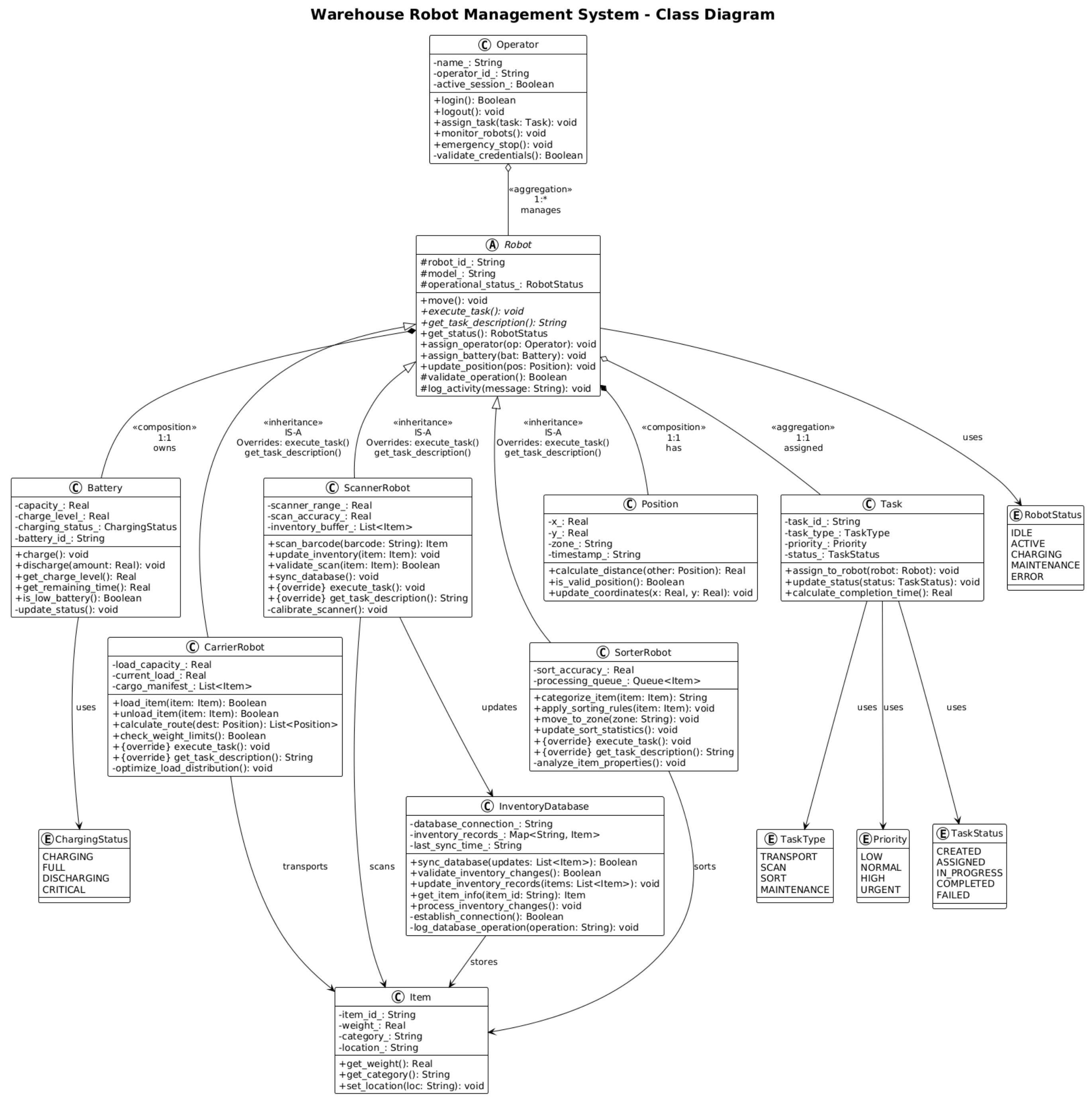
Warehouse Robot Management System - Class Diagram
Operator
-name_: String
-operator_id_: String
-active_session_: Boolean
+login(): Boolean
+logout(): void
+assign_task(task: Task): void
+monitor_robots(): void
+emergency_stop(): void
-validate_credentials(): Boolean
«aggregation» 1:* manages
Robot
#robot_id_: String
#model_: String
#operational_status_: RobotStatus
+move(): void
+execute_task(): void
+get_task_description(): String
+get_status(): RobotStatus
+assign_operator(op: Operator): void
+assign_battery(bat: Battery): void
+update_position(pos: Position): void
#validate_operation(): Boolean
#log_activity(message: String): void
«composition» 1:1 owns
«inheritance» IS-A Overrides: execute_task() get_task_description()
«inheritance» IS-A Overrides: execute_task() get_task_description()
«inheritance» IS-A Overrides: execute_task() get_task_description()
«composition» 1:1 has
«aggregation» 1:1 assigned
uses
Battery
-capacity_: Real
-charge_level_: Real
-charging_status_: ChargingStatus
-battery_id_: String
+charge(): void
+discharge(amount: Real): void
+get_charge_level(): Real
+get_remaining_time(): Real
+is_low_battery(): Boolean
-update_status(): void
ScannerRobot
-scanner_range_: Real
-scan_accuracy_: Real
-inventory_buffer_: List<Item>
+scan_barcode(barcode: String): Item
+update_inventory(item: Item): void
+validate_scan(item: Item): Boolean
+sync_database(): void
+{override} execute_task(): void
+{override} get_task_description(): String
-calibrate_scanner(): void
Position
-x_: Real
-y_: Real
-zone_: String
-timestamp_: String
+calculate_distance(other: Position): Real
+is_valid_position(): Boolean
+update_coordinates(x: Real, y: Real): void
Task
-task_id_: String
-task_type_: TaskType
-priority_: Priority
-status_: TaskStatus
+assign_to_robot(robot: Robot): void
+update_status(status: TaskStatus): void
+calculate_completion_time(): Real
RobotStatus
IDLE
ACTIVE
CHARGING
MAINTENANCE
ERROR
CarrierRobot
-load_capacity_: Real
-current_load_: Real
-cargo_manifest_: List<Item>
+load_item(item: Item): Boolean
+unload_item(item: Item): Boolean
+calculate_route(dest: Position): List<Position>
+check_weight_limits(): Boolean
+{override} execute_task(): void
+{override} get_task_description(): String
-optimize_load_distribution(): void
SorterRobot
-sort_accuracy_: Real
-processing_queue_: Queue<Item>
+categorize_item(item: Item): String
+apply_sorting_rules(item: Item): void
+move_to_zone(zone: String): void
+update_sort_statistics(): void
+{override} execute_task(): void
+{override} get_task_description(): String
-analyze_item_properties(): void
uses
updates
uses
uses
uses
ChargingStatus
CHARGING
FULL
DISCHARGING
CRITICAL
InventoryDatabase
-database_connection_: String
-inventory_records_: Map<String, Item>
-last_sync_time_: String
+sync_database(updates: List<Item>): Boolean
+validate_inventory_changes(): Boolean
+update_inventory_records(items: List<Item>): void
+get_item_info(item_id: String): Item
+process_inventory_changes(): void
-establish_connection(): Boolean
-log_database_operation(operation: String): void
TaskType
TRANSPORT
SCAN
SORT
MAINTENANCE
Priority
LOW
NORMAL
HIGH
URGENT
TaskStatus
CREATED
ASSIGNED
IN_PROGRESS
COMPLETED
FAILED
transports
scans
sorts
stores
Item
-item_id_: String
-weight_: Real
-category_: String
-location_: String
+get_weight(): Real
+get_category(): String
+set_location(loc: String): void

Figure 1: *Complete System Class Diagram*

4 System Workflows

4.1 Task Assignment Process

When an operator assigns a task, the system follows this workflow:

1. Operator calls `login()` to establish authenticated session.
2. Operator calls `assign_task()` with task specifications.
3. System creates new Task object with CREATED status.
4. System evaluates available robots based on:
   - Robot type compatibility with task type.

    - Current robot status (must be IDLE).
    - Battery level (must be sufficient for task).
    - Robot location relative to task location.
5. System selects optimal robot and calls `assign_to_robot()`.
6. Task status changes to ASSIGNED.
7. Robot receives task and calls `execute_task()`.
8. Task status changes to IN_PROGRESS.
9. Robot provides progress updates during execution.
10. Upon completion, task status changes to COMPLETED.
11. System logs completion and updates performance metrics.

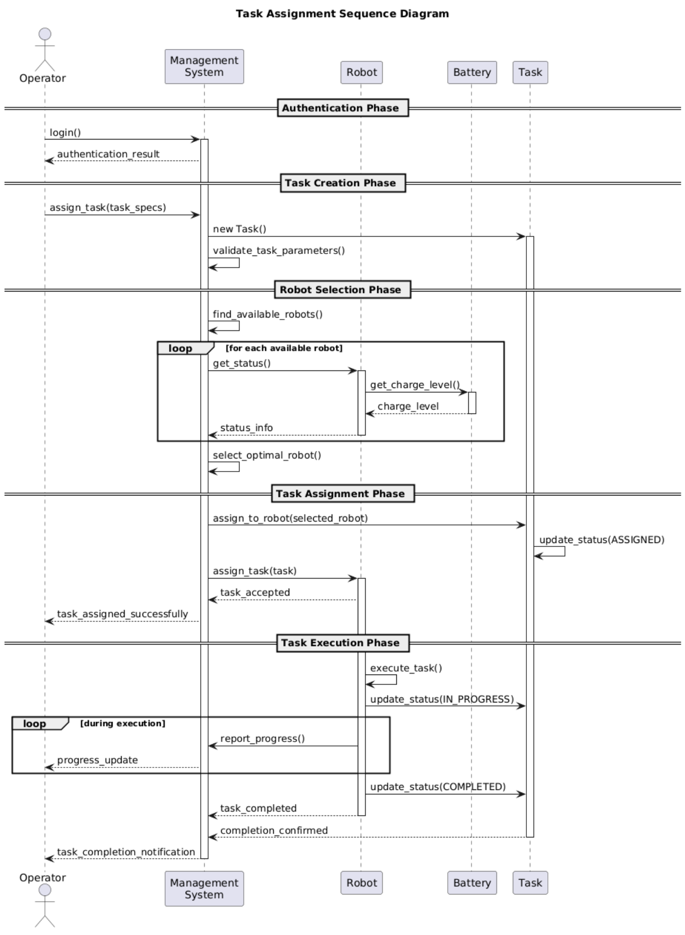

Figure 2: *Task Assignment Sequence Diagram*

4.2 Battery Management Process

Our system continuously monitors battery levels and automatically manages charging:

1. Robot continuously calls `get_charge_level()` during operations.
2. When battery calls `is_low_battery()` returns true:
    - Robot completes current task or safely interrupts if critical.
    - Robot status changes to CHARGING.
    - System finds available charging station.
    - Robot navigates to charging station.
3. Robot calls `charge()` to begin charging process.

4. Battery status changes to CHARGING.
5. System monitors `get_remaining_time()` for completion estimate.
6. When charging complete, battery status changes to FULL.
7. Robot status returns to IDLE and becomes available for tasks.

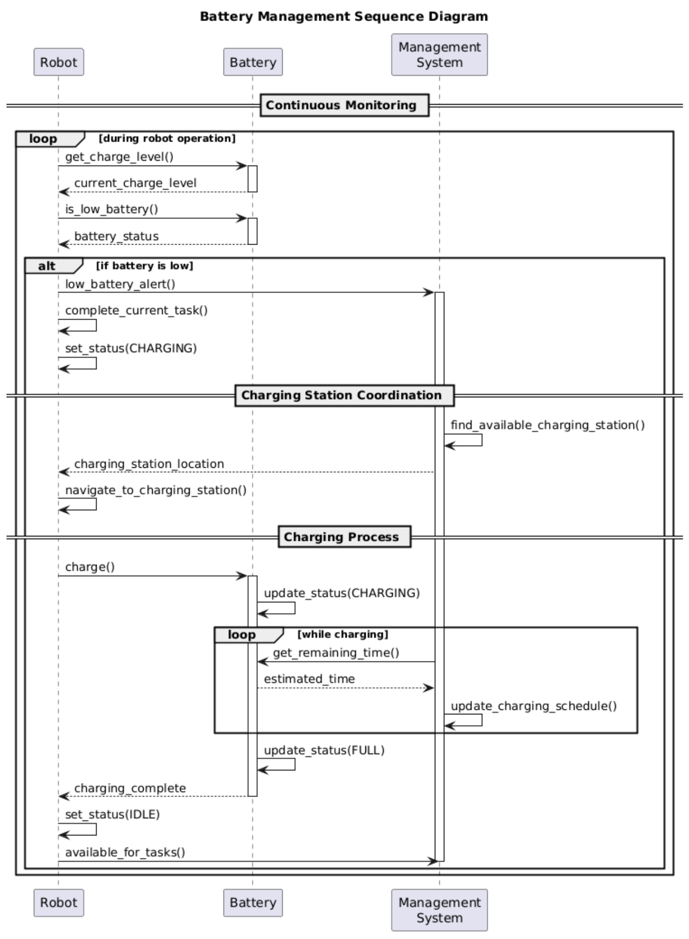

Figure 3: *Battery Management Sequence Diagram*

4.3 Robot-Specific Operations

4.3.1 CarrierRobot Transportation

When a CarrierRobot executes a transportation task:

1. `execute_task()` calls `calculate_route()` for optimal path.
2. Robot navigates to pickup location.
3. `load_item()` adds items to cargo with `check_weight_limits()` validation.
4. Robot updates cargo manifest and calls `optimize_load_distribution()`.
5. Robot navigates to destination following calculated route.
6. `unload_item()` removes items at destination.
7. Robot reports task completion.

4.3.2 ScannerRobot Inventory Scanning

When a ScannerRobot executes a scanning task:

1. `execute_task()` navigates to designated scanning zone.

2. Robot systematically scans items using `scan_barcode()`.
3. Each scan calls `validate_scan()` to ensure accuracy.
4. Valid scans are stored in inventory buffer.
5. `update_inventory()` processes scanned data.
6. `sync_database()` synchronizes with central inventory system.
7. Robot reports scanning completion with accuracy statistics.

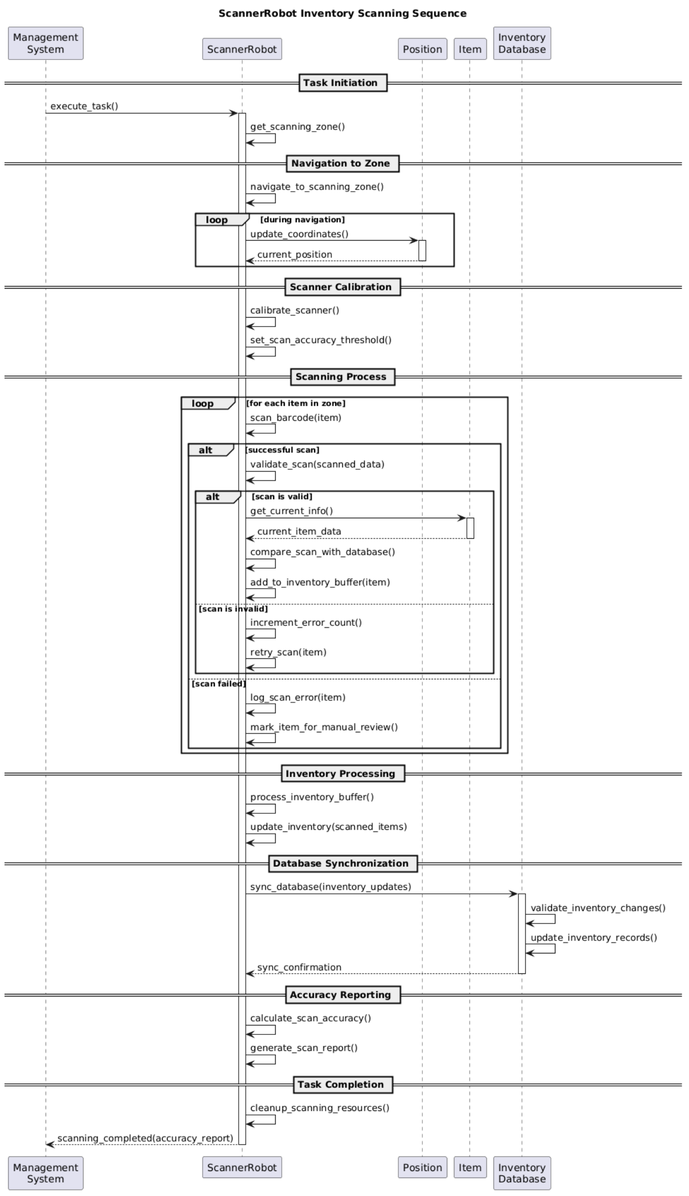

Figure 4: *Scanner Robot Sequence Diagram*

4.3.3 SorterRobot Package Organization

When a SorterRobot executes a sorting task:

1. `execute_task()` retrieves items from processing queue.
2. `analyze_item_properties()` examines item characteristics.
3. `categorize_item()` determines appropriate category.
4. `apply_sorting_rules()` validates sorting decision.
5. `move_to_zone()` transports item to designated area.

6. `update_sort_statistics()` records performance metrics.
7. Robot continues with next item in queue.

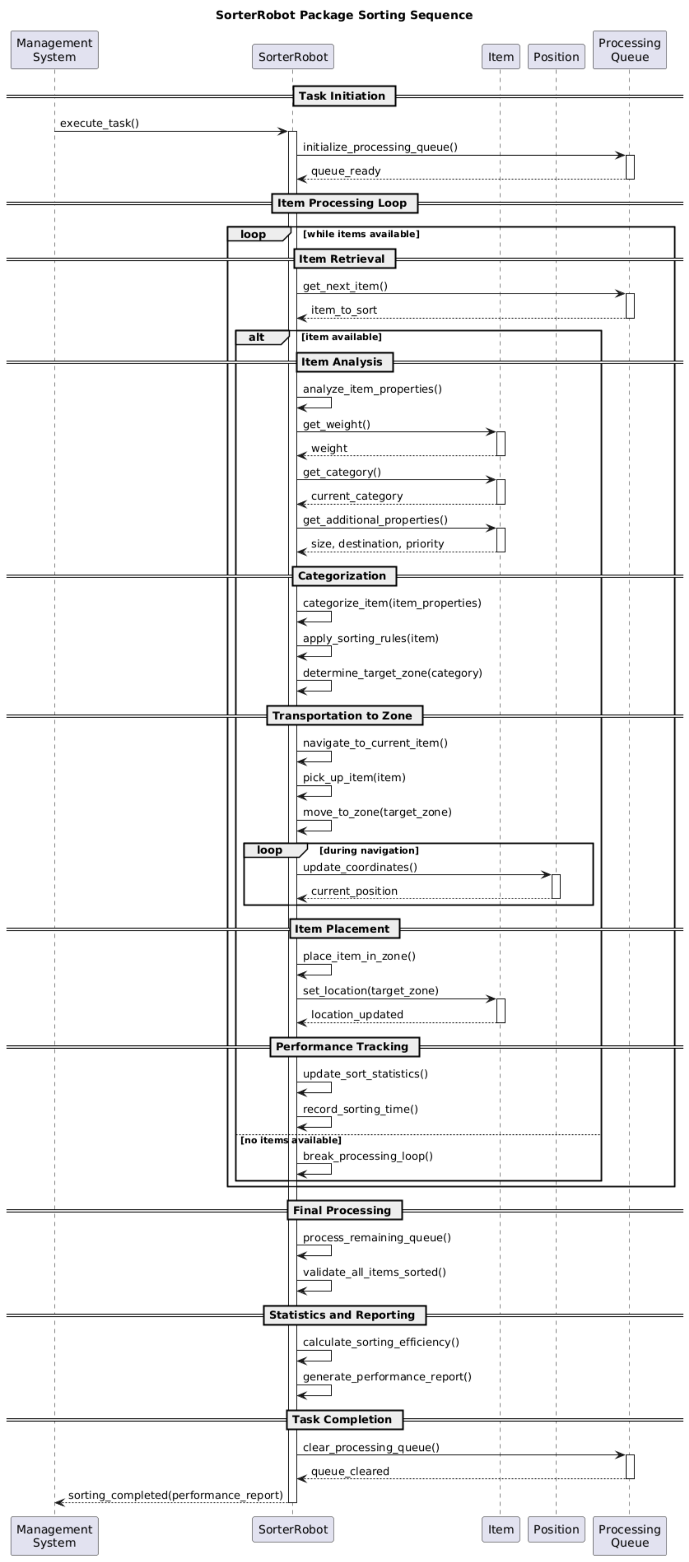

Figure 5: *Sorter Robot Sequence Diagram*

5 Implementation Guidelines

5.1 Development Phase 1: Core Infrastructure

Priority 1 - Abstract Base Classes:

- Implement Robot abstract base class with all method signatures.
- Create Battery class with full power management functionality.
- Implement Position class with coordinate tracking.
- Develop all enumeration types for status management.

Priority 2 - Basic Relationships:

- Establish Robot-Battery composition (each robot owns one battery).
- Establish Robot-Position composition (each robot owns position).
- Implement basic robot construction with battery and position assignment.

5.2 Development Phase 2: Robot Specializations

Implement each robot type in this order:

1. **CarrierRobot**: Start with basic load management and transportation.
2. **ScannerRobot**: Add barcode scanning and inventory integration.
3. **SorterRobot**: Complete with sorting logic and zone management.

For each robot type:

- Implement all specialized attributes and methods.
- Override abstract methods (`execute_task()`, `get_task_description()`).
- Test polymorphic behavior thoroughly.
- Validate inheritance relationships work correctly.

5.3 Development Phase 3: Management Systems

Operator Management:

- Implement Operator class with authentication.
- Establish aggregation relationship with robots.
- Create task assignment and monitoring capabilities.
- Add emergency stop functionality.

Task Management:

- Implement Task class with status tracking.
- Create task assignment and lifecycle management.
- Integrate with robot execution workflows.
- Add performance monitoring and reporting.

Database Management:

- Implement InventoryDatabase class with persistent storage capabilities.
- Establish database connection and synchronization protocols.
- Create data validation and integrity checking mechanisms.
- Integrate with ScannerRobot for inventory updates and synchronization.

5.4 Testing Strategy

Unit Testing Requirements:

- Test each class method individually with comprehensive edge cases.
- Verify all visibility modifiers are enforced correctly.

- Test abstract method implementations in all robot types.
- Validate enumeration usage and state transitions.

Integration Testing:

- Test composition relationships (robot-battery, robot-position).
- Test aggregation relationships (operator-robot, robot-task).
- Verify polymorphic method calls work across robot types.
- Test complete workflows from task assignment to completion.
- Test ScannerRobot and InventoryDatabase integration for data synchronization.
- Validate database connection management and error handling.

System Testing:

- Test multiple robots operating simultaneously.
- Verify battery management during concurrent operations.
- Test operator management of multiple robots.
- Validate emergency stop functionality across all robots.

6 Configuration and Deployment

6.1 System Configuration

Robot Configuration:

- `CarrierRobot`: Configure load capacity based on physical specifications.
- `ScannerRobot`: Set scanner range and accuracy requirements.
- `SorterRobot`: Define sorting accuracy and processing queue size.
- All robots: Set battery capacity and charging thresholds.

Warehouse Configuration:

- Define warehouse coordinate system and boundaries.
- Establish zone definitions for different areas.
- Configure charging station locations.
- Set up operator access controls and permissions.
- Configure database connection parameters and backup procedures.
- Establish inventory synchronization intervals and data validation rules.

6.2 Performance Considerations

Memory Management:

- Robot composition ensures automatic cleanup of Battery and Position.
- Use smart pointers for aggregation relationships where appropriate.
- Monitor memory usage with multiple concurrent robots.

Concurrency:

- Each robot operates independently with thread-safe status updates.
- Battery monitoring requires thread-safe charge level access.
- Operator commands must be thread-safe for multiple robot management.

This design provides a solid foundation for implementing a robust warehouse robot management system. The clear class hierarchies, well-defined relationships, and comprehensive workflows ensure that the system will be maintainable, extensible, and reliable in production use.