Design Phase Document

Robotics Competition Management System

Z. Kootbally[a]

[a]*MAGE, University of Maryland, College Park*

Abstract—This document captures the output of the design phase for the Robotics Competition Management System, a software tool for competition organizers and team managers. The system tracks team rosters, monitors robot status and battery levels, assigns and schedules tasks, models the arena layout, and records scoring. This document presents the domain model derived from noun/verb analysis, detailed class specifications with integrated business rules, structured behavioral scenarios with UML diagram references, and a phased implementation plan. The system is implemented across Lectures 6 and 7 of ENPM605.

Keywords—*OOP, design phase, robotics competition, Python, class design, UML, business rules, requirement analysis*

Contents

Robotics Competition Management System

1. System Overview

The Robotics Competition Management System is a software tool for competition organizers and team managers. It tracks team rosters, monitors robot status and battery levels, assigns and schedules tasks, models the arena layout, and records scoring. It does not implement robot control software (motor control, path planning, sensor fusion); it models the competition from the management perspective.

The system manages five core entities: robots, sensors, tasks, arenas, and teams. Robots are equipped with sensors and organized into teams. Tasks are assigned to individual robots, executed in an arena, and scored upon completion. The system enforces constraints on data validity (e.g., battery ranges, roster limits), triggers automatic state changes (e.g., low-battery alerts), and computes derived values (e.g., team scores).

Two supporting value types underpin the arena model. `Position` encodes a 2-D coordinate used to represent obstacle locations. `Rectangle` encodes an axis-aligned bounding box used to define named zones within the arena. Both are implemented as immutable frozen data classes; they carry no behavior of their own and are used exclusively as typed attribute values inside `Arena`.

This document serves as the bridge between the problem domain and the implementation phase, providing the blueprint from which Python code will be written in Lectures 6 and 7. Table 1 provides a quick reference for which classes and OOP concepts are covered in each lecture.

Table 1. Class-to-lecture mapping and OOP concepts introduced.

Lecture	Classes	OOP Concepts
L6	`Robot`, `Sensor`	Classes, objects, `__init` class attributes, dunder lation, `@property`
L7	`MobileRobot`, `ManipulatorRobot`, `Task`, `Team`, `Arena`, `Position`, `Rectangle`	Inheritance, polymorpl classes, composition, classes (frozen)

2. Domain Model

The domain model was derived from a two-step process. First, a noun/verb analysis of the competition description identified candidate classes and methods. Second, a specialization step examined the verbs attributed to robots and split the single `Robot` noun into an abstract base class and two concrete subclasses: `MobileRobot` (navigation) and `ManipulatorRobot` (manipulation).

2.1. Source Text

The following domain description was analyzed (nouns that became classes are shown in **bold blue**, candidate methods in *italic orange*):

> A robotics competition involves **teams** of **robots** collaborating to *complete* **tasks** in an **arena**. Each **robot** is equipped with **sensors** to *perceive* its environment. Some robots are **mobile robots** that *navigate* the arena at a given speed across different terrain types. Other robots are **manipulator robots** that *pick up* objects and *deliver* them to target zones using an extendable arm. Teams *register* robots, *assign* tasks based on priority, and *track* scores as tasks are completed. Each robot has a battery level that *drains* during task execution and can be *recharged* to full capacity.

Note

The original description uses only the generic term "robot." The noun/verb pass identifies two structurally distinct robot behaviors: navigation (speed, terrain) and manipulation (arm reach, payload). This is the trigger for the specialization step described in Section 2.4.

2.2. Extraction Results

The noun/verb analysis produced the following candidate classes. Note that this is a first-pass extraction; the class specifications in Section 3 refine these candidates by adding validation methods, computed properties, and implementation-specific signatures.

- **Robot** (abstract base class)
  - *Attributes:* `name`, `battery`, `status`, `tasks_completed`, `sensors`
  - *Methods:* `move()` (abstract — implemented by `MobileRobot` and `ManipulatorRobot`), `perform_task()`, `recharge()`, `assign_task()`
  - *Relationships:* Has `Sensors` (composition), belongs to `Team` (aggregation), assigned `Tasks` (association)
- **MobileRobot** (concrete subclass of `Robot`)
  - *Additional Attributes:* `max_speed`, `terrain_type`
  - *Methods:* `move()` (concrete implementation)
- **ManipulatorRobot** (concrete subclass of `Robot`)
  - *Additional Attributes:* `arm_reach`, `payload_capacity`
  - *Methods:* `move()` (concrete implementation), `pick_up()`, `deliver()`
- **Sensor**
  - *Attributes:* `sensor_type`, `range_m`, `accuracy`
  - *Methods:* `read()`, `calibrate()`
  - *Relationships:* Belongs to `Robot` (composition)
- **Task**
  - *Attributes:* `name`, `priority` (`int`), `duration_s`, `points`, `status`
  - *Methods:* `assign()`, `complete()`, `cancel()`, `is_complete()`
  - *Relationships:* Assigned to `Robot` (association)
- **Arena**
  - *Attributes:* `name`, `width`, `height`, `obstacles` (`list[Position]`), `zones` (`dict[str, Rectangle]`)
  - *Methods:* `is_clear()`, `get_zone()`, `add_obstacle()`
  - *Relationships:* Contains `Robots` (aggregation); uses `Position` and `Rectangle` as value types
- **Team**
  - *Attributes:* `team_name`, `robots`, `max_robots`, `score`
  - *Methods:* `add_robot()`, `remove_robot()`, `total_battery()`, `get_available_robots()`, `update_score`
  - *Relationships:* Has `Robots` (aggregation)
- **Position** (frozen data class)
  - *Attributes:* `x` (`float`), `y` (`float`)
  - *Methods:* auto-generated `__init__`, `__repr__`, `__eq__`, `__hash__`
  - *Relationships:* Used by `Arena` as the element type of `obstacles`
- **Rectangle** (frozen data class)
  - *Attributes:* `x` (`float`), `y` (`float`), `width` (`float`), `height` (`float`)
  - *Methods:* auto-generated `__init__`, `__repr__`, `__eq__`, `__hash__`
  - *Relationships:* Used by `Arena` as the value type of `zones`

"Competition" was considered during analysis and excluded; it may become a top-level orchestrating class in a future iteration. "Zones" and "obstacles" are not independent classes but are represented as typed collections inside `Arena` using the value types `Rectangle` and `Position` respectively.

2.3. Class Relationships

The following relationships were identified between classes. These directly inform how classes reference one another in the implementation.

- **Team to Robot (Aggregation):** A team manages a roster of robots. Robots can exist independently if the team is dissolved.
- **Robot to Sensor (Composition):** Sensors are owned by a robot. Destroying a robot destroys its sensors.
- **Robot to Task (Association):** A robot is assigned a task. Both exist independently; the assignment can be broken. `Robot` holds a `_current_task` reference; `Task` has no back-reference to `Robot`.
- **Arena to Robot (Aggregation):** The arena contains robots during competition. Robots exist independently of the arena.
- **Arena to Position / Rectangle (Dependency):** `Arena` uses `Position` as the element type of its `obstacles` list and `Rectangle` as the value type of its `zones` dictionary. Both are frozen data classes with no back-reference to `Arena`.
- **MobileRobot / ManipulatorRobot to Robot (Inheritance, L7):** Specialized robot types extend the abstract `Robot` base class and provide concrete implementations of `move()`.

2.4. Design Decisions

The noun/verb pass produces a flat list of candidate classes. Two additional design decisions were required before the class specifications could be written.

Specializing Robot into MobileRobot and ManipulatorRobot. The source text attributes two structurally incompatible behaviors to robots: navigation (verbs *navigate*, attributes speed and terrain type) and manipulation (verbs *pick up* and *deliver*, attributes arm reach and payload capacity). A single `Robot` class that carries all of these attributes would assign `None` to inapplicable fields for any given instance — a recognized design smell. The solution is specialization: `Robot` becomes an abstract base class that holds the shared interface (`name`, `battery`, `status`, `move()`), and the two concrete subclasses each add only what is unique to their type.

- **MobileRobot** adds `max_speed` and `terrain_type`; implements `move()` as direction-based navigation.
- **ManipulatorRobot** adds `arm_reach` and `payload_capacity`; implements `move()` and adds `pick_up()` and `deliver()`.

This is an instance of the **specialization** pattern introduced in L7: start from a general base class and push type-specific attributes and behaviors down into dedicated subclasses.

Introducing Position and Rectangle as value types. The `Arena` class stores two collections of structured data: a list of obstacle coordinates and a dictionary of named zone bounding boxes. Representing these as raw `tuple` objects would sacrifice type safety and readability. Instead, two frozen data classes are introduced — `Position(x, y)` and `Rectangle(x, y, width, height)` — that give names to the fields and enforce positivity constraints on `Rectangle` dimensions. Neither class carries behavior beyond what `@dataclass(frozen=True)` auto-generates.

3. Class Specifications

Each class specification includes its attributes with types and constraints, its methods with signatures and preconditions, and the business rules it enforces. Business rules are identified with a prefix: **BR-C** for constraints, **BR-T** for triggers, and **BR-D** for computations. Type annotations follow Python typing conventions.

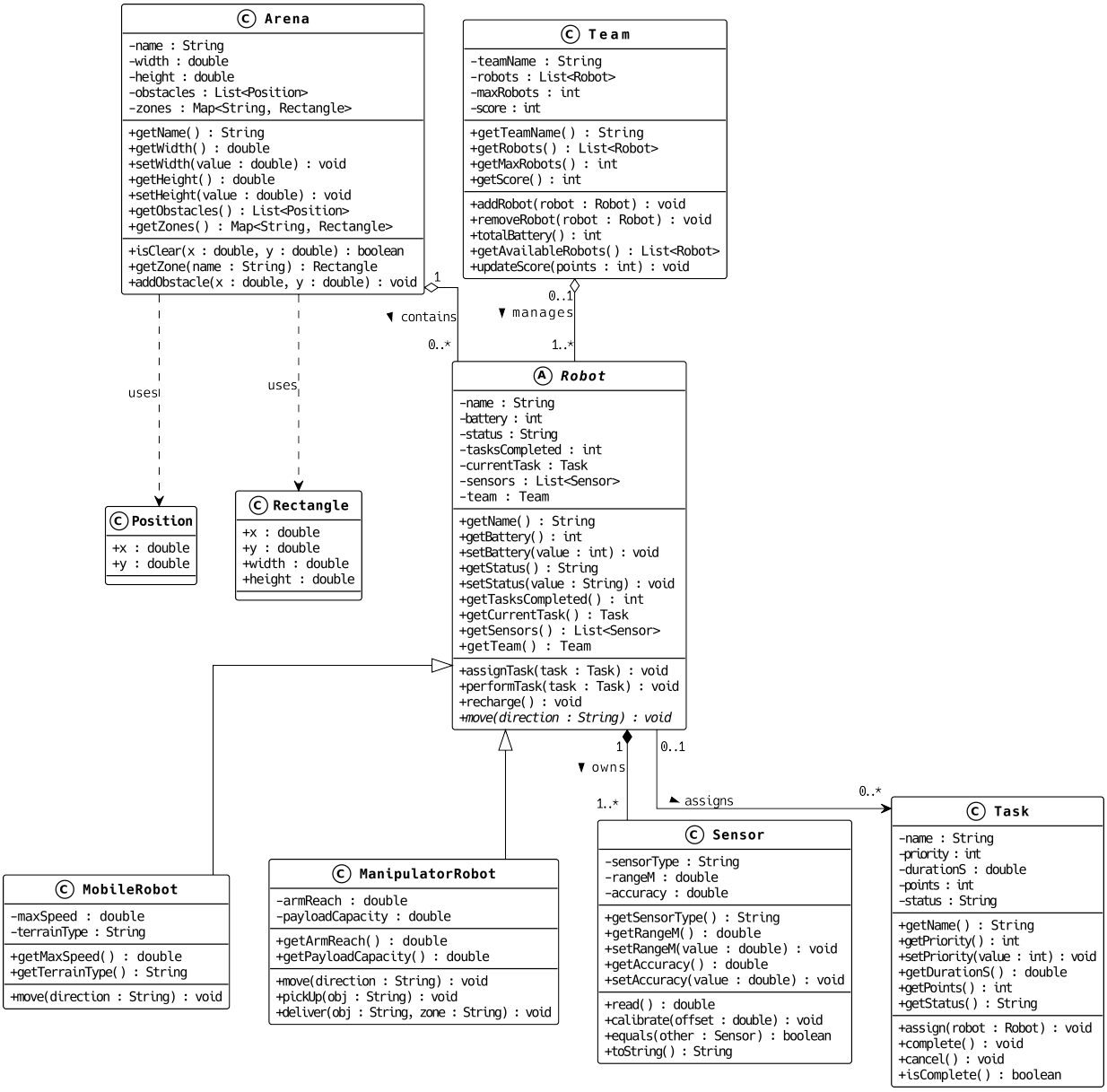

Figure 1. UML class diagram for the Robotics Competition Management System showing all classes with attributes, methods, and relationships.

Implementation details such as private helper methods and internal validation routines are deferred to the implementation lectures (L6 and L7). This section focuses on the public interface and the rules each class must enforce.

3.1. Robot

The central entity. Represents an autonomous agent registered to a team. `Robot` is an abstract base class; it cannot be instantiated directly. Concrete subclasses (`MobileRobot`, `ManipulatorRobot`) must implement `move()`.

3.1.1. Attributes

- **name** (`str`): Required. Read-only after creation.
- **battery** (`int`): Must be in range [0, 100]. Default: 100. Validated in the property setter.
- **status** (`str`): One of: `'active'`, `'busy'`, `'needs_recharge'`, `'offline'`. Default: `'active'`.
- **tasks_completed** (`int`): Non-negative. Incremented on task completion. Default: 0.
- **current_task** (`Task | None`): Reference to the currently assigned task. Default: `None`. Set via `assign_task()`.
- **sensors** (`list[Sensor]`): Composition. Sensors are created in the same expression as the robot and passed into the constructor. No external variable holds a reference to an individual sensor.

3.1.2. Methods

- **assign_task**`(task: Task) -> None`: Status must be `'active'`. Raises `RuntimeError` otherwise. Stores the task in `_current_task` and sets status to `'busy'`.
- **perform_task**`(task: Task) -> None`: Status must be `'busy'`. Drains battery by `task.duration_s * drain_rate`. Evaluates trigger rules after draining.
- **recharge**`() -> None`: Sets battery to 100. Sets status to `'active'`. Concrete method inherited by all subclasses.
- **move**`(direction: str) -> None`: **Abstract.** Must be implemented by every concrete subclass. Status must be `'active'` or `'busy'`.

3.1.3. Business Rules Enforced

- **BR-C01:** Battery must be an integer in [0, 100]. Enforced in the `battery` property setter.
- **BR-C02:** A robot cannot accept a task unless its status is `'active'`. Enforced as a precondition in `assign_task()`.
- **BR-T01:** When battery drops below 10, status automatically changes to `'needs_recharge'`. Evaluated after any operation that modifies battery.
- **BR-T03:** When battery reaches 0, status changes to `'offline'`. Evaluated after any operation that modifies battery.
- **BR-D01:** Battery drain = `task.duration_s * drain_rate`. Computed in `perform_task()`.

3.2. MobileRobot

A concrete subclass of `Robot` that moves across terrain. Provides a direction-based implementation of the abstract `move()` method. All other `Robot` methods are inherited without override.

3.2.1. Attributes

- **max_speed** (`float`): Must be positive. Maximum movement speed in m/s. Validated in the property setter.
- **terrain_type** (`str`): The terrain this robot is optimized for (e.g., `'flat'`, `'rough'`). Read-only after creation.

3.2.2. Methods

- **move**(`direction: str) -> None`: Concrete implementation of the abstract method inherited from `Robot`. Moves the robot one step in the given direction. Status must be `'active'` or `'busy'`.
- **assign_task**, **perform_task**, **recharge**: Inherited from `Robot` without override.

3.2.3. Business Rules Enforced

- Inherits BR-C01, BR-C02, BR-T01, BR-T03, BR-D01 from `Robot`.
- **BR-C09:** `max_speed` must be positive. Enforced in the `max_speed` property setter.

3.3. ManipulatorRobot

A concrete subclass of `Robot` that picks up and delivers objects. Provides a concrete implementation of `move()` and adds two manipulation-specific methods.

3.3.1. Attributes

- **arm_reach** (`float`): Must be positive. Maximum reach of the manipulator arm in meters. Validated in the property setter.
- **payload_capacity** (`float`): Must be positive. Maximum payload the arm can carry in kg. Validated in the property setter.

3.3.2. Methods

- **move**(`direction: str) -> None`: Concrete implementation of the abstract method inherited from `Robot`. Moves the robot one step in the given direction. Status must be `'active'` or `'busy'`.
- **pick_up**(`obj: str) -> None`: Picks up the named object at the robot's current position. Status must be `'busy'`.
- **deliver**(`obj: str, zone: str) -> None`: Delivers the named object to the specified zone. Status must be `'busy'`.
- **assign_task**, **perform_task**, **recharge**: Inherited from `Robot` without override.

3.3.3. Business Rules Enforced

- Inherits BR-C01, BR-C02, BR-T01, BR-T03, BR-D01 from `Robot`.
- **BR-C10:** `arm_reach` must be positive. Enforced in the `arm_reach` property setter.
- **BR-C11:** `payload_capacity` must be positive. Enforced in the `payload_capacity` property setter.

3.4. Sensor

Perception hardware mounted on a robot. Compared by range or accuracy.

3.4.1. Attributes

- **sensor_type** (`str`): One of: `'lidar'`, `'camera'`, `'ultrasonic'`. Read-only after creation.
- **range_m** (`float`): Must be positive. Validated in the property setter.
- **accuracy** (`float`): Must be in [0.0, 1.0]. Validated in the property setter.

3.4.2. Methods

- **read**`() -> float`: Returns `range_m` with simulated noise based on accuracy.
- **calibrate**(`offset: float) -> None`: Adjusts `range_m` by offset. Result must remain positive.
- **__repr__**`() -> str`: Returns `"Sensor('<type>', <range_m>)"`.
- **__eq__**(`other: Sensor) -> bool`: Two sensors are equal if their `range_m` values are equal.
- **__gt__**(`other: Sensor) -> bool`: Compares sensors by `range_m`.

3.4.3. Business Rules Enforced

- **BR-C06:** Sensor range must be positive. Enforced in the `range_m` property setter.
- **BR-C07:** Accuracy must be in [0.0, 1.0]. Enforced in the `accuracy` property setter.

3.5. Task

A unit of work assigned to a robot. Scored upon completion.

3.5.1. Attributes

- **name** (`str`): Required. Describes the task (e.g., "pick widget from zone A").
- **priority** (`int`): Must be a positive integer. Higher values indicate higher priority.
- **duration_s** (`float`): Must be positive. Estimated execution time in seconds.
- **points** (`int`): Must be non-negative. Points awarded on completion.
- **status** (`str`): One of: `'pending'`, `'in_progress'`, `'completed'`, `'cancelled'`. Default: `'pending'`.

3.5.2. Methods

- **assign**`(robot: Robot) -> None`: Status must be `'pending'`. Sets status to `'in_progress'`. Internally calls `robot.assign_task(self)`, which stores the task reference and sets the robot status to `'busy'`.
- **complete**`() -> None`: Status must be `'in_progress'`. Sets status to `'completed'`. Triggers team score update (BR-T02).
- **cancel**`() -> None`: Status must be `'pending'` or `'in_progress'`. Sets status to `'cancelled'`. Robot (if assigned) returns to `'active'`.
- **is_complete**`() -> bool`: Returns `True` if status == `'completed'`.

3.5.3. Business Rules Enforced

- **BR-C05:** Priority must be a positive integer. Enforced in the `priority` property setter.
- **BR-T02:** On task completion, team score is updated by adding `task.points`. Triggered in `complete()`.

3.6. Arena

The competition environment. Owns obstacle data and zone definitions. Arena behavior is primarily exercised in L7 scenarios involving robot navigation and task execution within the physical space. `Arena` uses two frozen data classes as value types: `Position` (Section 3.8) for obstacle coordinates and `Rectangle` (Section 3.9) for zone bounding boxes.

3.6.1. Attributes

- **name** (`str`): Required.
- **width** (`float`): Must be positive. Validated in the property setter.
- **height** (`float`): Must be positive. Validated in the property setter.
- **obstacles** (`list[Position]`): List of obstacle coordinates. Each element is a `Position(x, y)` frozen data class instance. Default: empty.
- **zones** (`dict[str, Rectangle]`): Named zones mapping a zone label to a `Rectangle(x, y, width, height)` bounding box. Default: empty.

3.6.2. Methods

- **is_clear**`(x: float, y: float) -> bool`: Returns `True` if the position `(x, y)` does not coincide with any obstacle in `obstacles`.
- **get_zone**`(name: str) -> Rectangle`: Returns the `Rectangle` bounding box for the named zone. Raises `KeyError` if the name is not found in `zones`.

- **add_obstacle**(x: float, y: float) -> None: Constructs a Position(x, y) and appends it to the obstacles list.

3.6.3. Business Rules Enforced

- **BR-C08:** Width and height must be positive. Enforced in property setters.

3.7. Team

A group of robots registered to compete. Manages roster and scoring.

3.7.1. Attributes

- **team_name** (str): Required. Read-only after creation.
- **robots** (list[Robot]): Managed via add/remove methods, not directly.
- **max_robots** (int): Must be positive. Maximum roster size. Default: 5.
- **score** (int): Non-negative. Updated on task completion. Default: 0.

3.7.2. Methods

- **add_robot**(robot: Robot) -> None: Roster must not exceed max_robots (BR-C04). Robot must not belong to another team (BR-C03). Raises ValueError on violation.
- **remove_robot**(robot: Robot) -> None: Robot must be in the roster. Raises ValueError otherwise.
- **total_battery**() -> int: Returns sum of battery across all robots (BR-D03).
- **get_available_robots**() -> list[Robot]: Returns robots with status == 'active'.
- **update_score**(points: int) -> None: Adds points to score. Called on task completion (BR-T02, BR-D02).

3.7.3. Business Rules Enforced

- **BR-C03:** Each robot belongs to at most one team at a time. Enforced in add_robot() by checking the robot's current team reference.
- **BR-C04:** Roster cannot exceed max_robots. Enforced as a precondition in add_robot().
- **BR-D02:** team.score = sum of completed task points. Maintained incrementally via update_score().
- **BR-D03:** total_battery() = sum of robot.battery for all robots. Computed on demand.

3.8. Position

A frozen data class representing a 2-D coordinate in the arena. Used as the element type of Arena.obstacles. Because it is frozen, instances are immutable and hashable; they can be stored in sets or used as dictionary keys.

3.8.1. Attributes

- **x** (float): Horizontal coordinate. No validation constraint; any finite float is accepted.
- **y** (float): Vertical coordinate. No validation constraint; any finite float is accepted.

3.8.2. Methods

All methods are auto-generated by @dataclass(frozen=True):

- **__init__**(x: float, y: float) -> None: Positional or keyword construction: Position(1.0, 2.0) or Position(x=1.0, y=2.0).
- **__repr__**() -> str: Returns Position(x=1.0, y=2.0).
- **__eq__**(other: object) -> bool: Field-by-field equality.
- **__hash__**() -> int: Stable hash based on field values. Available because frozen=True.

3.8.3. Business Rules Enforced

None. `Position` is a pure value type. Coordinate validity (e.g., staying within arena bounds) is the responsibility of `Arena`.

3.9. Rectangle

A frozen data class representing an axis-aligned bounding box in the arena. Used as the value type of `Arena.zones`. Immutable and hashable for the same reasons as `Position`.

3.9.1. Attributes

- **x** (`float`): Horizontal coordinate of the top-left corner.
- **y** (`float`): Vertical coordinate of the top-left corner.
- **width** (`float`): Must be positive. Width of the zone in arena units.
- **height** (`float`): Must be positive. Height of the zone in arena units.

3.9.2. Methods

All methods are auto-generated by `@dataclass(frozen=True)`, with validation in `__post_init__`:

- **`__init__`**`(x: float, y: float, width: float, height: float) -> None`: Positional or keyword construction.
- **`__post_init__`**`() -> None`: Validates that `width` and `height` are positive. Raises `ValueError` otherwise.
- **`__repr__`**`() -> str`: Returns `Rectangle(x=..., y=..., width=..., height=...)`.
- **`__eq__`**`(other: object) -> bool`: Field-by-field equality.
- **`__hash__`**`() -> int`: Stable hash based on field values.

3.9.3. Business Rules Enforced

- **BR-C12:** `width` must be positive. Enforced in `__post_init__`.
- **BR-C13:** `height` must be positive. Enforced in `__post_init__`.

4. Behavioral Scenarios

Each scenario documents a use case with preconditions, the main flow, postconditions, and exception paths. The two primary scenarios include references to sequence and activity diagrams.

4.1. UC-01: Assign a Task to a Robot

- **Actors:** Team manager (implicit, via code).
- **Preconditions:** Team has at least one robot with status `'active'`. Task exists with status `'pending'`.
- **Main Flow:**
  1. Manager calls `team.get_available_robots()` to find eligible robots.
  2. For a candidate robot, the system checks `robot.status == 'active'`.
  3. The system verifies `robot.battery` >= estimated drain for the task.
  4. `task.assign(robot)` is called, setting `task.status` to `'in_progress'` and internally calling `robot.assign_task(self)`, which stores the task reference and sets `robot.status` to `'busy'`.
- **Postconditions:** Task status is `'in_progress'`. Robot status is `'busy'`. Robot's `_current_task` references the task.
- **Exceptions:**
  - 3a. No robots have status `'active'`: raise `RuntimeError`.
  - 3b. Robot battery insufficient: skip to next candidate or raise error.

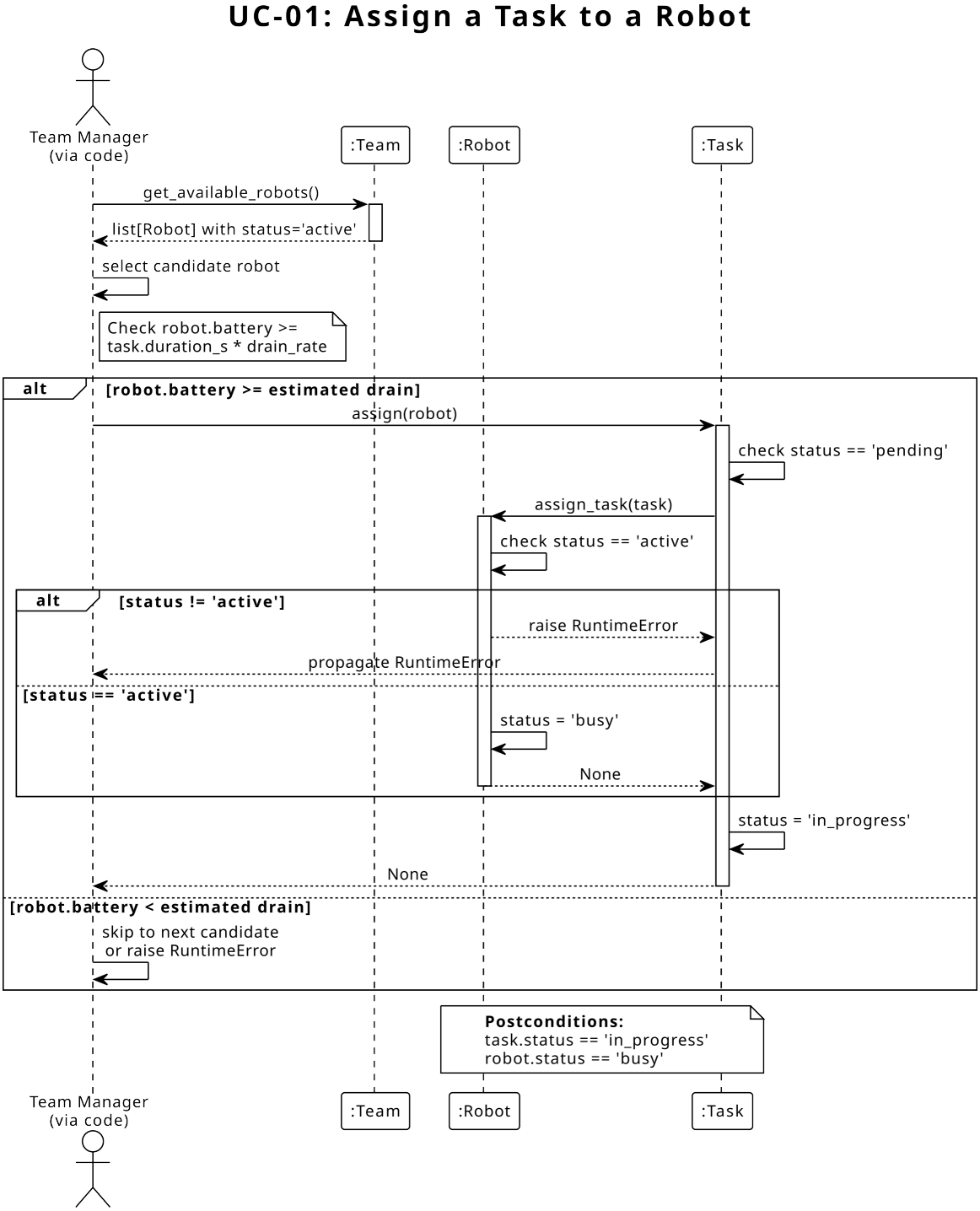

Figure 2. Sequence diagram for UC-01: Assign a Task to a Robot.

4.2. UC-02: Robot Completes a Task

- **Actors:** Robot (autonomous execution).
- **Preconditions:** Robot status is `'busy'`. A task with status `'in_progress'` is assigned.
- **Main Flow:**
  1. Robot checks battery level against `task.duration_s * drain_rate`.
  2. If sufficient: robot navigates to the target zone.
  3. Robot picks up the object at the source zone.
  4. Robot delivers the object to the target zone.
  5. Robot calls `task.complete()`, setting status to `'completed'`.
  6. `task.complete()` triggers `team.update_score(task.points)`.
  7. Robot increments `tasks_completed` and sets status to `'active'`.
  8. Battery trigger rules are evaluated (BR-T01, BR-T03).
- **Postconditions:** Task status is `'completed'`. Team score increased. Robot status is `'active'` (or `'needs_recharge'` if battery < 10).
- **Exceptions:**
  - 1a. Battery insufficient: robot calls `recharge()` first, then re-checks.
  - 4a. Battery depleted mid-task: task is cancelled, robot status set to `'offline'`.

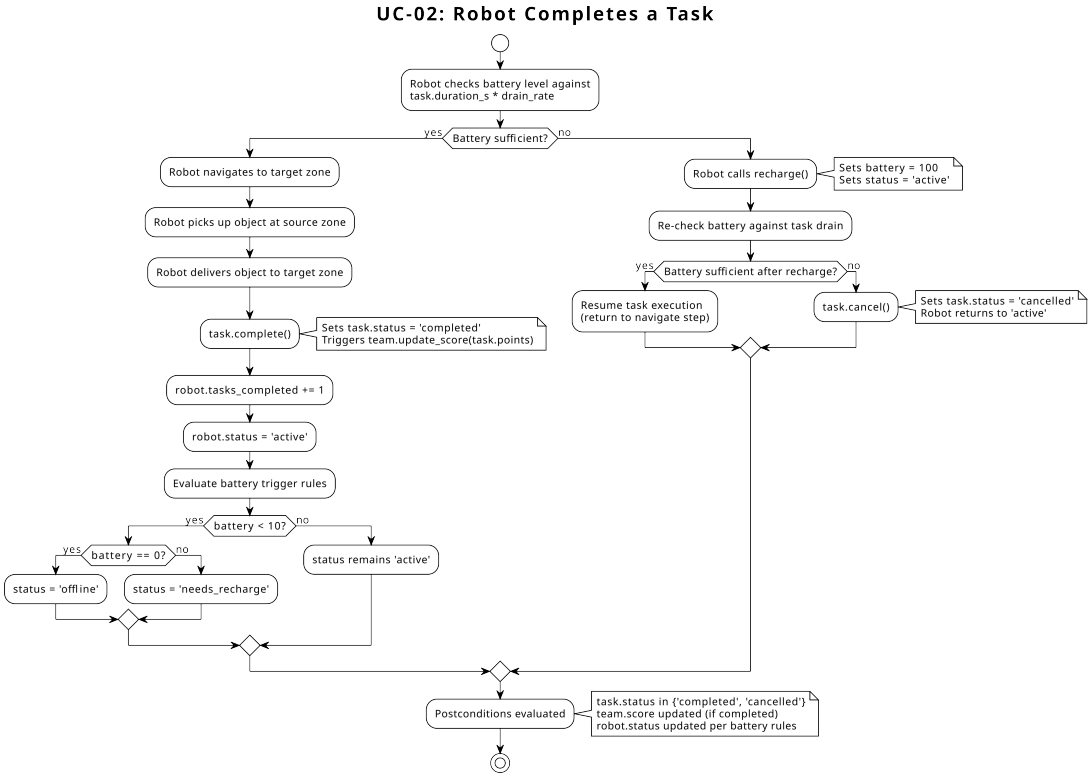

Figure 3. Activity diagram for UC-02: Robot Completes a Task.

4.3. UC-03: Robot Navigates the Arena (L7)

- **Actors:** Robot, Arena.
- **Preconditions:** Robot is assigned a task. Arena is initialized with obstacles and zones.
- **Main Flow:**
  1. Robot queries `arena.get_zone(target)` to obtain the target coordinates.

2. Robot checks `arena.is_clear(x, y)` along its planned path.
3. If the path is clear, robot moves to the target zone.
4. If an obstacle is encountered, robot recalculates the path.

- **Postconditions:** Robot has reached the target zone or raised an error.
- **Exceptions:**
  - 1a. Zone not found: `KeyError` is raised.
  - 2a. No clear path exists: robot reports failure.

4.4. Additional Use Cases

- **UC-04: Register a Team.** Create a `Team` with a name and `max_robots`. Postcondition: empty roster, score = 0.
- **UC-05: Add Robot to Team.** Call `team.add_robot(robot)`. Enforces BR-C03 (unique team) and BR-C04 (roster limit).
- **UC-06: Equip Robot with Sensor.** Create `Sensor` and append to `robot.sensors`. Enforces BR-C06 and BR-C07.
- **UC-07: Recharge a Robot.** Call `robot.recharge()`. Sets battery to 100, status to `'active'`.
- **UC-08: Cancel a Task.** Call `task.cancel()`. Status must be `'pending'` or `'in_progress'`. Robot (if assigned) returns to `'active'`.
- **UC-09: Query Team Standings.** Call `team.score` and `team.total_battery()`. Returns current score and aggregate battery.

5. Implementation Plan

5.1. Project Structure

Each class lives in its own module. This promotes modularity and makes the code easier to maintain, test, and extend.

- `robot.py`: `Robot` abstract base class (L6)
- `sensor.py`: `Sensor` class (L6)
- `mobile_robot.py`: `MobileRobot` class (L7)
- `manipulator_robot.py`: `ManipulatorRobot` class (L7)
- `task.py`: `Task` class (L7)
- `team.py`: `Team` class (L7)
- `arena.py`: `Arena` class (L7)
- `position.py`: `Position` frozen data class (L7)
- `rectangle.py`: `Rectangle` frozen data class (L7)
- `main.py`: Entry point (L6 / L7)

5.2. Phased Implementation

Table 1 (Section 1) summarizes the mapping. The two phases are:

- **L6: Core Classes.** `Robot` (abstract base class) and `Sensor` with full validation, dunder methods, and `@property`. Students implement classes from the specifications in Sections 3.1 and 3.4, learning encapsulation and operator overriding.
- **L7: Relationships and Specialization.** `MobileRobot`, `ManipulatorRobot`, `Task`, `Team`, `Arena`, `Position`, and `Rectangle`. Students provide concrete implementations of the abstract `Robot` interface, model composition, aggregation, and polymorphism, and practice frozen data classes for typed value objects. Specifications in Sections 3.2 through 3.3 and 3.5 through 3.9.

5.3. Business Rules Traceability

Every business rule identified in the class specifications maps to a specific enforcement point in the code.

Constraints:

- **BR-C01:** Battery in [0, 100]. Enforced in the `Robot.battery` property setter.
- **BR-C02:** Robot must be `'active'` to accept tasks. Enforced in `Robot.assign_task()` precondition.
- **BR-C03:** Robot belongs to at most one team at a time. Enforced in `Team.add_robot()` check.
- **BR-C04:** Roster cannot exceed `max_robots`. Enforced in `Team.add_robot()` precondition.
- **BR-C05:** Priority must be a positive integer. Enforced in the `Task.priority` property setter.
- **BR-C06:** Sensor range must be positive. Enforced in the `Sensor.range_m` property setter.
- **BR-C07:** Accuracy in [0.0, 1.0]. Enforced in the `Sensor.accuracy` property setter.
- **BR-C08:** Arena dimensions must be positive. Enforced in the `Arena` width/height property setters.
- **BR-C09:** `MobileRobot.max_speed` must be positive. Enforced in the `max_speed` property setter.
- **BR-C10:** `ManipulatorRobot.arm_reach` must be positive. Enforced in the `arm_reach` property setter.
- **BR-C11:** `ManipulatorRobot.payload_capacity` must be positive. Enforced in the `payload_capacity` property setter.
- **BR-C12:** `Rectangle.width` must be positive. Enforced in `Rectangle.__post_init__`.
- **BR-C13:** `Rectangle.height` must be positive. Enforced in `Rectangle.__post_init__`.

Triggers:

- **BR-T01:** Battery < 10 triggers `'needs_recharge'`. Evaluated after any operation that modifies battery.
- **BR-T02:** Task completion triggers score update. Enforced in `Task.complete()` which calls `team.update_score`
- **BR-T03:** Battery == 0 triggers `'offline'`. Evaluated after any operation that modifies battery.

Computations:

- **BR-D01:** Drain = duration × rate. Computed in `Robot.perform_task()`.
- **BR-D02:** Score = sum of task points. Maintained incrementally via `Team.update_score()`.
- **BR-D03:** Total battery = sum across roster. Computed on demand in `Team.total_battery()`.

6. Glossary

- **Abstract Base Class (ABC):** A class that cannot be instantiated directly and is intended to be subclassed. Defined using Python's `ABC` from the `abc` module. May contain both abstract methods (which subclasses must override) and concrete methods (inherited as-is). `Robot` is the ABC in this system.
- **Aggregation:** A "has-a" relationship where the part can exist independently of the whole (e.g., `Team` has `Robots`; `Arena` contains `Robots`).
- **Association:** A general relationship between two classes where neither owns the other (e.g., `Robot` is assigned a `Task`).
- **Business Rule:** A constraint, trigger, or computation that the system must enforce as part of domain logic.
- **Composition:** A "has-a" relationship where the part cannot exist without the whole (e.g., `Robot` has `Sensors`).
- **Concrete Class:** A class that provides implementations for all abstract methods it inherits and can therefore be instantiated directly. `MobileRobot` and `ManipulatorRobot` are the concrete subclasses of `Robot` in this system.
- **Domain Model:** The set of classes, attributes, methods, and relationships that represent the problem space.
- **Drain Rate:** A constant factor determining how much battery is consumed per second of task execution.
- **Dunder Method:** A Python method with double underscores (e.g., `__init__`, `__repr__`) that enables operator overriding and integration with built-in functions. Implementing a dunder method in a subclass

is an instance of method overriding, not overloading.

- **Encapsulation:** Bundling data and methods together while restricting direct access to internal state.
- **Frozen Data Class:** A `@dataclass(frozen=True)` instance whose fields cannot be reassigned after construction. Python generates `__hash__` automatically, making frozen instances usable as dictionary keys and set members. Used in this system for `Position` and `Rectangle`.
- **Noun/Verb Analysis:** A technique for extracting candidate classes (nouns) and methods (verbs) from a natural-language description.
- **Position:** A frozen data class with fields `x` and `y` representing a 2-D coordinate in the arena. Used as the element type of `Arena.obstacles`.
- **Precondition:** A condition that must be true before a method can execute. Raises an exception if violated.
- **Property:** A Python mechanism (via the `@property` decorator) that allows controlled access to an attribute through getter and setter methods while preserving attribute-style syntax. Used throughout this system to enforce validation on assignment.
- **Rectangle:** A frozen data class with fields `x`, `y`, `width`, and `height` representing an axis-aligned bounding box. Used as the value type of `Arena.zones`.
- **UML:** Unified Modeling Language. A standardized notation for visualizing software design.
- **Value Type:** An object whose identity is defined entirely by its field values rather than by reference. Two value type instances with identical fields are considered equal. In this system, `Position` and `Rectangle` are value types implemented as frozen data classes.