System Implementation

1. Overview and alignment with Unit 7

This implementation realises the humanoid robot specified in Unit 7. The codebase maps directly to the activity, class, sequence, and state diagrams, with minor updates during development to improve cohesion and testability. Core behaviour—navigation, manipulation, communication, power and charging management, and error recovery—is encapsulated in a layered, object-oriented design with clear seams for testing and evolution. This submission addresses the learning outcomes by providing UML-backed design, appropriate data structures, and efficient search algorithms.

Repository layout

- `robot/domain/`: core entities and contracts (`RobotState`, `Waypoint`, `EnvObject`, `Environment`, `MemoryStore`, `PathPlanner`).
- `robot/services/`: pluggable planning strategies (`AStarPlanner`, `GreedyPlanner`), a `Navigator` orchestrator, a simple `EventBus`, and actuator adapters (`Manipulator`, `Communicator`).
- `robot/interface/cli.py`: a minimal command queue used by the interactive CLI.
- `robot/controller.py`: the `Robot` façade coordinating state guards, auto-dock/charging, and command processing.
- `robot/tests/`: unit tests.
- `__main__.py`: enables `python -m robot`.

This organisation supports the brief's emphasis on best practice, documentation, PEP-8 style, and test evidence.

2. Object-oriented design and patterns

- **Strategy** — `PathPlanner` is implemented by `AStarPlanner` and `GreedyPlanner`. `Navigator` composes a planner, enabling algorithm substitution without controller changes (Gamma et al., 1995).
- **Observer (lightweight)** — `EventBus` provides decoupled publish/subscribe; tests exercise this to demonstrate extensibility for telemetry and UI callbacks.
- **(Planned) State** — `Robot.tick` currently collects most control-flow. A State refactor (e.g., `OffState`, `IdleState`, `MovingState`, `ManipulatingState`, `ChargingState`, `ErrorState`) is planned to reduce branching and localise invariants.

These choices reduce coupling and localise decision density, aiding maintainability and testing.

3. Data structures and algorithms

Entities & structures. `Waypoint` is hashable and totally ordered; `Environment` maintains objects, an obstacle map and sensor readings; `MemoryStore` records action breadcrumbs; `deque` backs step queues. **Planning & navigation.**

- **A***: `AStarPlanner` maintains `g_score` and a min-heap (`heapq`). A mild obstacle-aware penalty nudges the search away from blocked cells. Returns a `deque[(x, y), …]` or `None`.
- **Greedy**: deterministic step-towards-target with obstacle checks and a bounded iteration counter.
- **Navigator**: composes a planner, stores the step queue, and exposes a test-friendly `navigate()` that streams to `_drive_to()` (Spy pattern in tests).

4. Robot controller behaviour

Power transitions (`power_on`, `power_off`), guards for charging/docking, auto-dock trigger when battery < 10%, charging progression on `tick`, and error recovery (`ERROR→IDLE` when battery ≥ 10%). Deterministic “no path” scenarios via a `timeout_counter`. Exceptions in `navigate`/`pick` are logged and surfaced as user-readable errors.

5. Testing and automated evidence

The suite under `robot/tests` uses `unittest` **and** direct `assert` statements (as required) to cover: normal/obstacle navigation, timeouts/no-path, manipulation success/failure, charging lifecycle, guards (off/busy/charging), EventBus behaviour, and planner polymorphism.

Artifacts (linked):

- Unit test log · Pylint · Radon CC · Radon MI

Snapshot (6 October 2025). Cyclomatic Complexity: 132 analysed blocks (A=120, B=7, C=4, E=1); Maintainability Index average ≈ 81 (A). Pylint ≈ 9.9/10. The sole E-rank hotspot is `controller.py:tick` (CC=38).

6. Style and documentation (PEP-8/257)

Module/class/method docstrings, type hints, consistent naming. Tests keep concise helpers; any linter suppressions are targeted and test-only. A `.pylintrc` and reproducible commands regenerate evidence (Alchin, 2010).

7. Structure and presentation

The layered layout (domain/, services/, interface/, `controller.py`) simplifies change management, supports unit testing via seamful interfaces, and is ready for packaging (e.g., `pyproject.toml` with a CLI entry-point).

8. Limitations and planned improvements

- **Control-flow concentration:** planned State refactor to reduce CC and clarify invariants.
- **Environment realism:** extend map handling/sensing; add profiling.
- **Events:** extend `EventBus` for telemetry/metrics.

9. How to run and reproduce evidence

Create a virtual environment and install tools

python -m venv .venv

.\.venv\Scripts\pip install -U pip pylint radon

Run the CLI

python -m robot

Run tests and capture evidence

python -m unittest discover -s robot\tests -p "test_*.py" -v > docs\tests.txt

.\.venv\Scripts\python -m pylint robot > docs\pylint.txt

.\.venv\Scripts\python -m radon cc -s -a robot > docs\radon_cc.txt

.\.venv\Scripts\python -m radon mi -s robot > docs\radon_mi.txt

10. References (Harvard)

Alchin, M. (2010) *Pro Python*. Apress. Chidamber, S.R. and Kemerer, C.F. (1994) ‘A metrics suite for object-oriented design’, *IEEE TSE*, 20(6), pp. 476–493. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. McCabe, T.J. (1976) ‘A complexity measure', *IEEE TSE*, SE-2(4), pp. 308–320. Romano, F. and Kruger, H. (2021) *Learn Python Programming: An In-Depth Introduction to the Fundamentals of Python*. 3rd edn. Packt.