Reflection OOP_PCOM7E e-Portfolio

(Rolfe et al., 2001: What – So What – Now What; method note below)

Method (how I reflected)

I used **Rolfe et al. (2001)** to structure the narrative (What / So What / Now What). To deepen criticality, I applied prompts from the **University of Edinburgh Reflectors' Toolkit (2024)** to surface emotions, alternatives, and consequences, and cross-checked academic integrity and presentation against the **University of Essex (2024)** reflective-practice guidance.

What — Learning Journey and Key Developments

Before this module, my modelling experience was limited to GUI tools such as Lucidchart or draw.io.
Writing **UML as code with PlantUML (PUML)** was entirely new.
Treating `.puml` diagrams as first-class, version-controlled artefacts changed my mindset: models became diff-able, reviewable, and refactorable alongside source code.
Each diagram – class, sequence, activity, state – mapped directly to `robot_system.py`, closing the gap between design intent and implementation.

The *Humanoid Robot* artefact anchored learning across Units 8 to 12. I progressed from sketching behaviour to **executable designs** supported by tests and metrics. Early results showed **coverage 88 %**, **Maintainability Index 68**, one **Radon E-rank** hotspot in the controller, and **Pylint 8.6 / 10**.
Through iterative refactors, dependency inversion (Strategy pattern) and clearer state guards, the final metrics reached **coverage 95 %**, **MI 81**, **no E/F ranks**, and **Pylint 10.00 / 10**.

PUML kept documentation honest: every code change forced a diagram update, preventing architectural drift.

An industry parallel that clarified my design choices was **Oracle's modular cruise-management stack**, where discrete subsystems – guest handling, safety, cashless accounting – coordinate without tight coupling.
That separation-of-concerns lens validated *Strategy now* and *State later* for my robot controller.
In particular, Oracle's **Cruise Shipboard PMS/Fleet Documentation** underscored the value of **stateful workflows and compatibility matrices** for scalable decision-making, which I mirrored in my PUML and test plans (Oracle, 2024a; Oracle, 2024b).

Critical Incidents (with emotions, decisions and outcomes)

1) Static-analysis wall (Unit 10)
Emotion: anxiety and irritation when Pylint flagged dozens of issues after a "working" build.
Decision: pause feature work and adopt a lint-then-code rhythm with CI gates.
Outcome: warnings fell from 47 → 0; functions gained docstrings; readability improved. More importantly, I internalised that style enforces shared team cognition, not bureaucracy.

2) E-rank complexity in `tick` (Unit 11)
Emotion: overwhelm and hesitation.
Decision: split responsibilities and plan a State-pattern refactor (Idle / Moving / Manipulating / Charging / Error).
Outcome: complexity dropped E → C; defects during extension testing fell 3 → 0.

3) Model drift between PUML and code (Unit 9 → 11)
Emotion: irritation and mild embarrassment when I had to update the same sequence diagram repeatedly because it no longer matched reality.
Decision: I reminded myself that I value **quality over quantity**, so I deliberately forced consistency between design and implementation – even when it felt tedious. I committed to maintaining this discipline in my professional career.
Outcome: the diagrams regained explanatory power for reviewers, and onboarding for new developers became faster and more intuitive.

So What — Critical Analysis, Knowledge and Limits

Linking to Learning Outcomes

LO1 – Appraise and evaluate information-systems concepts
I moved from assuming inheritance to challenging it. Composition plus Strategy reduced coupling, and the guarded state machine clarified safety behaviour. Reading metrics through **McCabe (1976)** and **Chidamber & Kemerer (1994)** reframed quality as quantified risk reduction rather than aesthetic preference.

LO2 – Design or modify and document an OO system
PUML as code enabled traceable documentation: each public method has a diagram locus, enforcing design-to-code alignment.

LO3 – Develop in Python / SQL
The Strategy planner and dependency inversion allowed planner substitution without touching `Robot`. Test doubles made behaviour deterministic, lifting coverage beyond 95 %.

LO4 – Develop, implement and evaluate solutions for business decisions
Quality gates and metrics (MI, CC, coverage) created decision-ready evidence for risk-based release choices.

Challenging orthodoxy (originality)

Classic OOP teaching defaults to inheritance hierarchies. In practice, my initial `RobotBase` subclass design increased coupling and reduced testability. **Composition** plus interfaces proved simpler and safer, aligning with the Law of Demeter.
Patterns can also be overused: a premature Observer added needless indirection, so I reverted to a single Strategy and postponed State until metrics justified it. Patterns must *earn their keep* through measurable improvements.

This stance resonates with **explainability-first OO approaches** such as **X-OODM** (Javed et al., 2024), which advocate composition and state models that make decision paths auditable at design time. Bringing that lens to robotics strengthened my insistence on “patterns that earn their keep” and PUML-as-code to surface rationale for reviewers.

Emotional analysis (deeper)

I noticed a cycle of **irritation** → **curiosity** → **flow**. Irritation arose when tools exposed weaknesses in my habits; naming the emotion turned it into a hypothesis (“why is this hard?”).
That metacognitive shift converted frustration into structured experimentation and sustained motivation throughout the module.

Application to Professional Practice

I ported the portfolio’s CI gates into my DevOps consultancy pipeline: lint violations per PR dropped significantly and mean review time dropped also significantly.
The reflective habit – measure → reflect → refactor – scaled beyond the assignment. A “diagram-of-record” rule now guides production code reviews; new developers report faster comprehension within the first week.

Now What — Concrete and Measurable Next Steps

1) Complete State-pattern refactor
Target: no function > CC 10; maintain Pylint ≥ 9.8, MI ≥ 75, no E/F ranks.

2) Energy-aware profiling study
Replicate Şanlıalp et al. (2022) with modern tooling and extend using recent evidence on energy impact of design patterns (Poy et al., 2024).
Compare Strategy-only vs Strategy+State on identical workloads; record CPU / energy and correlate with MI / CC deltas to evidence sustainability claims.

3) Evidence-first PR template
Add a metrics table (delta of CC, coverage, MI) and link to PUML diff in every pull request to make risk and intent explicit.

4) Knowledge sharing
Deliver an internal session on “PUML as living spec”, capturing Q&A to refine team guidelines.

Conclusion

This portfolio shows a shift from feature-first coding to **evidence-led, sustainable OOP**.
Critical incidents and emotional growth mattered as much as algorithms: naming frustration, quantifying risk, and iterating deliberately produced better software and habits.
Challenging inheritance orthodoxy, deferring patterns until metrics justified them, and embracing PUML-as-code collectively delivered safer, clearer designs.

Going forward I will evaluate architecture by **measurable deltas (complexity, coverage, MI, energy)** and keep design alive in code and PUML.

Measure objectively – reflect critically – refactor sustainably – repeat.

References (Harvard)

Ackerman, E. (2024) ‘Humanoid robots are getting to work’, *IEEE Spectrum*. Available at: https://spectrum.ieee.org/humanoid-robots (Accessed: 14 October 2025).

Chidamber, S.R. and Kemerer, C.F. (1994) ‘A metrics suite for object-oriented design’, *IEEE Transactions on Software Engineering*, 20(6), pp. 476–493. Available at: https://ieeexplore.ieee.org/document/295895 (Accessed 6 October 2025).

Javed, A., Shoaib, M. and Jaleel, A. (2024) ‘X-OODM: Explainable Object-Oriented Design Methodology’, *IEEE Access*, 12, pp. 150807–150823. Available at: https://ieeexplore.ieee.org/document/10713364 (Accessed: 8 October 2025).

McCabe, T.J. (1976) ‘A complexity measure’, *IEEE Transactions on Software Engineering*, SE-2(4), pp. 308–320. Available at: https://ieeexplore.ieee.org/document/1702388/ (Accessed: 8 October 2025).

Oracle (2024a) *Oracle Hospitality Cruise Shipboard Property Management System – Installation Guide, Release 20.3*. Available at: https://docs.oracle.com/en/industries/hospitality/cruise/property_management/ (Accessed: 14 October 2025).

Oracle (2024b) ‘Cruise software and technology’. Available at: https://www.oracle.com/hospitality/cruise/ (Accessed: 14 October 2025).

Poy, O., Moraga, M.A., García, F. and Calero, C. (2024) ‘Impact on energy consumption of design patterns, code smells and refactoring techniques: a systematic mapping study’. Available at: https://www.sciencedirect.com/science/article/abs/pii/S0164121224003479 (Accessed: 14 October 2025).

Romano, F. and Kruger, H. (2021) *Learn Python Programming*. 3rd edn. Packt.

Rolfe, G., Freshwater, D. and Jasper, M. (2001) *Critical Reflection in Nursing and the Helping Professions: A User’s Guide*. Basingstoke: Palgrave Macmillan.

Şanlıalp, İ., Öztürk, M.M. and Yiğit, T. (2022) ‘Energy efficiency analysis of code refactoring techniques for green and sustainable software in portable devices’, *Electronics*, 11(3), p. 442. doi: 10.3390/electronics11030442.

University of Edinburgh (2024) ‘Reflecting on experience’, *Reflectors’ Toolkit*. Available at: https://reflection.ed.ac.uk/reflectors-toolkit/reflecting-on-experience (Accessed: 14 October 2025).

University of Essex (2024) ‘Reflective practice’, *Study Skills*. Available at: https://www.essex.ac.uk/student/study-skills/reflective-practice (Accessed: 14 October 2025).