ePortfolio Submission

> **Please find the ePortfolio for the module under https://essex.sevale.de/modules/oop/eportfolio/.**

Module reflection

Before starting this module, I already had solid experience with object-oriented programming and worked daily with systems built around OOP principles. So I wasn't expecting major changes in how I write code or structure classes. What the module did offer, though, was a chance to slow down, organize my thoughts, and put names to some of the practices I already instinctively use. In a few cases it also made me reconsider approaches I had previously dismissed or overlooked.

The main assignment of the module — designing and implementing a humanoid waiter robot — gave a practical way to look at object-oriented design from the ground up. The first part focused on the system design and required the use of UML diagrams: activity, sequence, state transition, and class diagrams. At first, this felt like an academic task. I've used UML design and generation tools before, but never in a structured way: creating UML from scratch, especially using standalone tools, always felt slow and a bit outdated. Following UML's specification and methodology to the letter evokes the notion of applying the rigid Waterfall model for software development, where there is indeed a dedicated design stage (Agarwal *et al.*, 2023). This doesn't seem to match the pace or culture of most modern software teams that tend to prefer the more flexible Agile approach (Hoda, Salleh and Grundy, 2018), where the support of the heavyweight UML diagrams would be seen as a burden. Still, I ended up appreciating some parts of it more than I expected. The state transition diagram, in particular, made the system's behavior much clearer. It wasn't about following diagramming rules — it was about seeing how the system could behave from a higher level. Visualizing robot states and request types helped me spot how different actions can happen in parallel and understand how tasks like delivering food and handling payments might overlap in a real implementation. That level of clarity would have taken longer to get through just code.

So even though I still don't see UML as widely practical, I appreciated the benefits from spending some time on modeling. Before, when I'd sketch things out quickly in a tool or on paper I wouldn't call it a separate stage in the software development process. Now I see modeling as a structured way to explore and test some ideas before writing code. In larger projects, using the right level of modeling early on can help avoid problems later. Besides, with the help of academic sources I articulated other benefits and goals of the modeling process such as discovering the different aspects of real-world objects to later express them in code (Isoda, 2001).

The second part of the module, building the system in Python, was more familiar. The logic and structure were clear, and I applied standard object-oriented techniques like encapsulation and polymorphism. Coming from languages like Kotlin, I did notice Python's dynamic typing made some parts feel less predictable. The logic wasn't hard, but I spent more time double-checking types and looking up syntax than I normally would. In future versions of the module, I think giving students the option to choose from a few OOP languages would make sense, especially for those who already have experience.

Even with the minor challenges, the implementation part had some important takeaways. The main one was the role of automated testing. I've written tests before, but starting from scratch this time made their value clearer. Writing tests for robot behaviors, task handling, and billing helped me catch problems early and made it easier to refactor later. The tests didn't just meet the requirements — they became a reference for how the system was supposed to work. In a few cases, I used them to understand the impact of a change more quickly than I could by reading the code. Unlike diagrams, which can become outdated, the tests remained useful.

Another part of the module I found helpful was the discussion forums. Having to explain my thoughts on topics like software complexity metrics or modeling for object-oriented systems helped me reflect more carefully. Reading other students' posts gave me some new angles on things I thought I already understood. It also made the module feel more connected to real development work, not just coursework.

One concept that stood out to me was software complexity measurement. I already knew that clean and well-structured code is easier to maintain, but I hadn't explored metrics like cyclomatic complexity, fan-in, or the CK/MOOD suites in detail (McCabe, 1976; Abreu and Carapuça, 1994; Chidamber and Kemerer, 1994). Seeing how these metrics try to quantify aspects like cohesion and coupling was interesting. I'm not planning to chase numbers in my own code, but having a clearer language to describe design problems is useful.

One of the more satisfying moments in the module was setting up PlantUML and seeing it generate UML diagrams from code. I hadn't used UML this extensively before — aside from the exposure in the previous module — so it was nice to discover a coding-oriented tool that made the process simple and efficient. It was less about the visual output and more about how seamlessly it fit into a developer's workflow.

Looking back, the module didn't change the way I write code, but it did help me better understand the patterns I already use. It also pushed me to think more intentionally about design decisions. For someone with prior experience, I'd suggest skipping the basic Python sections if possible, but definitely engaging with the testing, complexity, and modeling parts. Even if you will never use UML again, understanding when and why it can help is still valuable.

The main thing I'll take away is the broader idea of modeling — not necessarily using formal UML, but the act of giving rough shape to a system before building it. It's worth doing even in a quick or informal way. Whether it's a sketch, a diagram, or a block of pseudocode, modeling helps surface problems early and brings clarity. And sometimes that five-minute diagram can save hours of debugging later.

References

Abreu, F.B. and Carapuça, R. (1994) ‘Object-oriented software engineering: Measuring and controlling the development process’, in *Proceedings of the 4th international conference on software quality*, pp. 1–8.

Agarwal, Archi, Agarwal, Avni, Verma, D.K., Tiwari, D. and Pandey, R. (2023) ‘A Review on Software Development Life Cycle’, *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 9(3), pp. 384–388. Available at: https://doi.org/10.32628/CSEIT2390387.

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://doi.org/10.1109/32.295895.

Hoda, R., Salleh, N. and Grundy, J. (2018) ‘The Rise and Evolution of Agile Software Development’, *IEEE Software*, 35(5), pp. 58–63. Available at: https://doi.org/10.1109/MS.2018.290111318.

Isoda, S. (2001) ‘Object-oriented real-world modeling revisited’, *Journal of Systems and Software*, 59(2), pp. 153–162. Available at: https://doi.org/10.1016/S0164-1212(01)00059-0.

McCabe, T.J. (1976) ‘A Complexity Measure’, *IEEE Transactions on Software Engineering*, SE-2(4), pp. 308–320. Available at: https://doi.org/10.1109/TSE.1976.233837.