Object-Orientated Information Systems

Project Reflection

Unit 7: Design

Module 2 dealt with introducing students to building and documenting an information system. Unit 7 focused on delivering UML class, activity and state diagrams using the knowledge gained in prior units. Since I have experience developing UML diagrams, I considered it relatively simple to achieve the outcome. However, I was (painfully) surprised to realise that this was not so. The process I took was first to identify the nouns and verbs of the use case scenario presented. From there, I then extracted each statement that referenced a noun or a verb into a unique requirement entry. I did this so that I could ensure the output diagrams met *every* requirement put forth.

From the formulated list of requirements, I then built the UML class diagram. At first, it *was* relatively simple to develop. However, each requirement challenged my original design ideas. I often critically analysed my thoughts against discussions in the forum about UML relationships, especially the book by Fowler (2013). Mixed with my knowledge of relationships from ArchiMate modelling—that association is the *simplest, a catch-all* relation—this design was undoubtedly challenging and enjoyable.

The time taken to iron out correct relationships for the UML class diagram took close to two full weeks to complete. However, I settled on an approach (evidenced in unit 7) by testing the class structure, their inheritance, and associations, and ensuring they meet each requirement. Once agreed, I then use the class diagrams to drive the activity diagram. And I used the activity diagram to model the state diagram. All was well until the model required some written description. This once more challenged design assumptions, and I felt I could still do better; but time was not on my side, and I had to complete and submit the assignment. I enjoyed developing the activity diagram and learnt about a new activity type (interruptible) that is used to model an activity that can be abruptly interrupted. This was an enriching experience to know this.

After submission, I was disappointed in the grade received and felt that specific feedback points were silly, such as using a branch/fork in the activity diagram. And, comparing the feedback against work experience, I consider business analysts would typically resolve the issue. They would do so by adding/amending the missing requirement (not stated in the use-case)—Agile methodology. Another point was the feedback that the submission did not make use of references. I contemplated that unit 7 is wholly about UML *diagramming*; how could I reference another person's academic work for this assignment? Overall, I remain displeased that massive effort resulted in a paltry reward (an unsatisfactory grade).

Unit 11 Implementation

In unit 11, we focused on turning the UML diagram into Python code using object-orientated programming techniques. This activity was rather enjoyable, but it was unclear what type of interaction (if any) is required to show a working solution. So, I relied on work experience to guide me and set about building a collection of Python classes, ensuring that each method is correctly commented (as stated in the requirements for submission). In terms of style, I considered Python style code from Van Rossum et al. (2001). According to Bafatakis et al. (2019), the most common violation of Python code style relates to bad whitespace, accounting for over 34% of style violations. They note that "On average, posts with fewer violations per statement are favoured by the community".

While developing the Python code, an intriguing thought arose *private* and *protected* instance variables. In Python, every member in a class is *public* by default; there is no default private/protected mechanism like languages such as Java and C#. To resolve this understanding, I researched and uncovered the convention: name mangling via double underscores. Developers use double underscores for private members (Van Rossum & Drake, 2003) and a single underscore for protected members.

When I set about writing the tests to demonstrate the system end-to-end, I realised that Python does not have the notion of *typecasting* between *custom class* types. Though, Python does support typecasts between integral types. I also considered type-hinting in Python and its support for generic types. At first, I made the mistake of representing instance variables as class variables. But, writing tests highlighted this error, which I resolved by moving instance variable declarations into the initialiser method of each class. I chose to build two tests: one to demonstrate the activity diagram and another to demonstrate the state diagram. I was most pleased with the activity diagram test because it showed the code model concerning the UML class diagram from unit 7.

Lastly, I developed a README file using Markdown. I chose Markdown because it is the most straightforward text editor without relying on OpenOffice or Microsoft Word editors. In the document, I listed changes made to the code that deviated from the UML model, assumptions about the project, and functionality. Overall, I think the presentation of such a README file is of great benefit to developers who wish to understand the code and how it fits together.

References

Bafatakis, N., Boecker, N., Boon, W., Salazar, M.C., Krinke, J., Oznacar, G. & White, R. (2019) Python coding style compliance on stack overflow. *IEEE/ACM 16th International Conference on Mining Software Repositories* 1:210-214.

Fowler, M. (2013) UML Distilled: a brief guide to the standard object modeling language. 3rd ed. Boston, MA: Addison Wesley

Van Rossum, G., Warsaw, B. & Coghlan, N. (2001) PEP 8: style guide for Python code. Available from http://cnl.sogang.ac.kr/cnlab/lectures/programming/python/PEP8_Style_Guide.pdf. [Accessed 23 Jul 2021]

Van Rossum, G. & Drake Jr, F.L. (2003) *An Introduction to Python release 2.2.2*. Bristol: Network Theory Limited.