**Reflections on my journey of learning Object-Oriented Programming (OOP)**

The introduction to Python programming and Object-Oriented Programming (OOP) has been an eye-opener for me, especially considering that my background is in a non-computer science field. I have realized how OOP has fundamentally changed my approach to coding.

In Unit 1, I learned the basics of object-oriented programming, its major features, abstract classes, applying inheritance, using encapsulation, and polymorphism. Initially, the concept of classes and objects was a bit abstract and challenging to grasp. To augment my understanding, I had to take additional Python courses online. As I delved deeper into the subject, I began to understand the power and flexibility that OOP offers.

One of the first concepts I learned was the idea of a 'class' - a blueprint for creating objects. I found it fascinating how classes allowed for encapsulation, where data and methods could be bundled together. This not only made the code more organized but also enhanced its reusability. Since I was using a lot of online resources, the interchangeable use of terms such as properties, attributes, and fields was initially confusing. Similarly, methods, functions, and operations were interchangeably used in my readings. I managed to get more clarity through further readings.

Inheritance was another concept that stood out to me in OOP. The ability to create a new class, a 'child', from an existing 'parent' class, while inheriting its attributes and methods, was a game-changer. It allowed for code to be reused and extended, reducing redundancy.

**Key lessons in Unit 2**

Object-Oriented Analysis (OOA) is typically conducted in an iterative and incremental way. The outputs of OOA activities are analysis models. The intention is for these to be continuously refined and evolved, driven by key factors like risks and business value1.

While there isn’t a single hard and fast methodology for conducting OOA, there are several well-established approaches and practices. These include:

**Problem Domain Understanding**: During OOA, the software engineer works closely with domain experts and stakeholders to understand the problem that the software is intended to solve. This involves gathering requirements, identifying stakeholder’s needs, and understanding the business processes.

**Use Case Modeling**: Use cases are used to define the interactions between various actors (people, other systems, etc.) and the system. OOA uses use cases to help developers understand the requirements of a system and to design software systems that meet those requirements.

**Unified Modeling Language (UML)**: UML is a standardized notation for creating diagrams that represent different aspects of a software system. OOA uses UML diagrams to represent the different components and interactions of a software system.

**Design Patterns**: Design patterns are reusable solutions to common problems in software design. OOA uses design patterns to help developers create more maintainable and efficient software systems.

**Key lessons from Unit 3 (Week 3)**

During the 3rd week, we covered Unified Modelling Language (UML). I did not find UML to be an entirely new concept, despite my background being in non-computer science field. I actually used UML several times before without knowing much about the required standardization and how one is different from another type. I managed several software development project recently where I prepared and used ‘use-case diagrams’ and ‘activity diagrams’ as part of the scope of work for the software developers. In my understanding back then, these ‘diagrams’ are just a creative way of explaining your desired software to the developer. In a way – I thought that the ‘diagrams’ are very much subjective and there are no rules and standards to follow which is quite the opposite of what I learnt in Unit 3.

UML is a standardized modeling language used in object-oriented programming (OOP). It provides an integrated set of diagrams to help system and software developers specify, visualize, construct, and document the artifacts of software systems. UML is not a programming language itself, but rather a visual language used to represent the design of a system.

UML diagrams can be broadly classified into three main groups:

1. Behavior diagrams: These diagrams represent the dynamic behavior of the objects in a system, which can include their methods, interactions, and activities.
2. Interaction diagrams: These diagrams depict the interactions between objects and the order in which these interactions occur.
3. Structure diagrams: These diagrams show the static structure of the system using objects, attributes, operations, and relationships.

UML is widely used in the software development process and is an important part of developing object-oriented software.

Different types of UML diagrams are used at different stages of the Software Development Life Cycle (SDLC). Here’s a brief overview:

1. Use Case Diagrams: Used during the requirements gathering and high-level design phase to identify common sets of activities that users of the system indulge in. They help in understanding the functional requirements of the system.
2. Class Diagrams: Used as early as the high-level design process to define the domain model for the application, specifically, the relationship of data objects within the system, the relationships between them, and the operations that they can perform.
3. Sequence Diagrams/Communication Diagrams: Known collectively as interaction diagrams, they are used during the high- and low-level design process to show the interactions between objects in the system. They are also very helpful in the testing state when creating testing processes and procedures.
4. Activity Diagrams: Used during requirements gathering and high-level design to further identify process flows within the system.
5. Object Diagrams: These provide a snapshot of a system at an instance in time and are used to model complex structures.

The choice and use of these diagrams can vary depending on the specific needs of the project and the development methodology being used.

**Key lessons from Unit 4**

In Unit I had a chance to explore further on the differences of each UML types. I used Draw.io application for the drawings. With continuous practice, I learnt more about the differences of each UML type and how they are used at the different stages of SDLC. While referring to online literature one thing that caught my attention is the variations of each UML type. For example, you would notice how one author applies use case diagram much differently from another. In some cases, some big variations had me ask – where is the standardization?

**Key learning from Unit 5 (week 5)**

Continued practicing more on UML while also brainstorming on the summative assessment for designing UML for a python OOP proposal for driverless car. I conducted online literature search around the same topic. I must say – there is a lot to learn about the domain of driverless car than UML and OOP itself.

**Key learning from Unit 6 (week 6)**

As someone who has always wondered and got so curios as to how software programmers manage to break highly complex functions into codes, the lessons in unit 5 and 6 provided some answers. Polymorphism stood out for me as a fundamental concept in OOP that allows for greater efficiency, flexibility, and simplicity in code. Together with the other pillars of OOP this, I understand, is one of the reasons why programmers can handle very complex programs with relative ease.

**Key learning from Unit 7**

In week 7, I spent much time preparing for the summative assessment - a design proposal of software to support operation of driverless car. As I indicated above, the assignment requires a lot of reading to understand the technical domain of a driverless car because of the complexity of the technology. Unfortunately, it left me with less time to organize my understanding from the online searches and to apply the concept using python programming. I will cover more reflection on this under Unit 11.

Apart from the preparation and submission of the summative assessment, debugging, data structures and data search were covered in Unit 7. Debugging or error handline is not a new concept at all for me with a little coding experience that I had in the past. However, learning how Python handles errors did add a lot to my understanding of the concept. A key concept that I learnt about error handling in Python is how it is done through the use of exceptions that are caught in try blocks and handled in except blocks. Exceptions are errors that have correct syntax but generate an error when running the syntax. A good example here is a division by zero. If an error is encountered, a try block code execution is stopped and transferred down to the except block. In addition to using an except block after the try block, you can also use the finally block. The code in the finally block will be executed regardless of whether an exception occurs (Beckles & Dowling, 2012)

**Key learning from Unit 8**

Data structures were partly covered in Unit 7, but the practical aspects were covered in Unit 8. I do recall we covered data structures in the previous module as well. Here is a summary from my readings on how data structures play a crucial role in supporting Object-Oriented Programming (OOP).

1. **Organizing Data**: Data structures provide a way to organize data in an efficient and meaningful manner. They define the relationship between different pieces of data and specify ways of accessing, storing, and organizing the data (Goodrich et al, 2013)..
2. **Supporting OOP Principles**: Data structures can be used to implement key OOP principles. For instance, a class (a fundamental concept in OOP) can be seen as a data structure that holds data (attributes) and functions (methods) that operate on the data (Goodrich et al, 2013).
3. **Enhancing Code Efficiency and Flexibility**: Efficient use of data structures can lead to more efficient and flexible code. For example, using an appropriate data structure like a hash table can significantly speed up data retrieval operations (Goodrich et al, 2013).
4. **Implementing Real-World Entities:** In OOP, real-world entities are represented as objects. These objects are instances of classes, which can be thought of as user-defined data structures. These classes encapsulate data and the methods that operate on them(Goodrich et al, 2013).
5. **Supporting Abstraction and Encapsulation**: Data structures in OOP (like classes) support the principle of abstraction (showing only necessary details)and encapsulation (binding data and functions together), making it easier to manage complexity (Goodrich et al, 2013).

**Key learning from Unit 9**

Unit 9 was about packaging and testing – which provided an essential insight on how I could package and test my python OOP code for the upcoming submission for summative assessment. Eventually, I decided to package my codes into three modules according to my three selected functions to support a driverless car.

**Key learning from Unit 10**

In Units 9 and 10, we covered packaging, testing, and testing code in practice. I found the lecture cast in unit 9 very informative and simple to understand, especially on the Cyclomatic complexity. The lessons from these units also provided me with adequate insights on how I could apply the tools and approaches to test the quality of my python code for the upcoming summative assessment – implementation of object-oriented python program to support driverless car. Based on these insights, I applied automated unit testing, functional testing, and structural testing in my codes. Below is a screenshot of the unit testing that I applied in my implementation of the driverless car software using the in-built unittest module of python.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

The code provided above is implementing unit tests for two classes: LaneDetector and ObjectDetection. It uses Python’s built-in unittest module to generate numpy array for the LaneDetector and outputting visualization for the ObjectDetection class.

In addition to the unit testing, I also conducted function testing by importing sample images for both lane detection and object detection, while also conducting a structural test to check how the codes are set up using the OOP standards.

**Key learning from Unit 11**

In Unit 12, I dedicated much of my time to finalizing the submission for summative assessment – Implementation of Python OOP for driverless car. As I described earlier, understanding the technology domain of driverless car on its own requires quite a lot of time, let alone application of the concept using python (especially for a beginner like me). Besides, almost all of the online literature sources make references to various libraries that are used with python for developing a driverless car software. Some of these libraries are OpenCV, TensorFlow, TensorFlow hub, numpy, matplotlib, Carla, L5kit, YOLO5/3, and PythonRobotics. I could not ignore the libraries because they seemed to be essential to executing the code. For example, graying the picture colour, applying the Gaussian Blur and Hough-Lines required CV2.

Nonetheless, I still faced several frustrations during the coding and testing process of my project.

1. Importing of the libraries which worked well in my Visual Studio Code turned out to be different when transferring to Codio for submission. Either Codio did not support some of them or there were version compatibility issues.
2. Even in Visual Studio Code –installation of l5kit was not successful; and I could not test the third module (Navigation and Control).

Despite the two challenges above I am quite happy with what I have learnt and the experience I gained throughout the coding process.

Overall, learning OOP has been a transformative experience. It has not only equipped me with a powerful tool for problem-solving but also changed how I think about code organization and reusability. As I continue my coding journey, I look forward to exploring more advanced OOP concepts and applying them in my future projects.

**References**

Beckles, B. and Dowling, B. (2012). Python 3: Handling errors. [online] Available at: https://help.uis.cam.ac.uk/system/files/documents/exceptions.pdf [Accessed 16 Feb. 2024].

Goodrich, M.T., Tamassia, R. and Goldwasser, M.H. (2013) *Data Structures and algorithms using Python*. Available at: https://nibmehub.com/opac-service/pdf/read/Data Structures and Algorithms Using Python.pdf (Accessed: 15 February 2024).

‌