1 Introduction

This report details the solutions achieved using a procedural programming approach and an object-oriented approach to programme the shop assignment as part of the Multi-Paradigm Programming module.

1.1 Folder structure & files explained

This section of the report details the folder structure and provides a brief explanation of the files included in the folder. The main project folder is titled:

- "G00387815_Keith_Quinn_Shop_Assignment".

Within the main folder there are two sub folders:

1. Shop_in_C (ignore this folder for this report).
2. Shop_in_Python (only this folder is considered in this report).

Within the Shop_in_Python folder there are 2 files:

1. shop_OOP.py
2. shop_procedural.py

These are the two files that are of interest for this report. In the next section, both the object-oriented programming (OOP) and the procedural implementations of the shop assignment are discussed.

Also included in the main project folder are the following 5 csv files (these files are used in both implementations):

1. stock.csv (file used to stock the shop).
2. A.csv (customer A file to demonstrate when the shop hasn't enough stock).
3. B.csv (customer B file to demonstrate when the customer hasn't enough money).
4. C.csv (customer C file to demonstrate when the order can be fully processed).
5. Live.csv (file used in live mode to take customer and order details, overwritten for each new customer).

2 Object Oriented and Procedural Implementations

Both the OOP and the procedural implementations use the same menu to provide identical “user experience” as per the assignment specification. Figure 1 shows the menu that is displayed when either implementation is started.

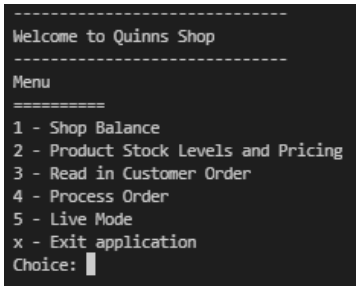

Figure 1 OOP and procedural "Menu"

When making comparisons between OOP and procedural paradigms one of the fundamental differences is that in OOP the class of the object has both state and functionality, unlike procedural which only has state. Consider Figure 2 and Figure 3 below

```
@dataclass # customer dataclass with name, budget and shopping list
class Customer:
    name: str = ""
    budget: float = 0
    shopping_list: List[ProductStock] = field(default_factory=list)
```

Figure 2 Customer class in procedural

```
class Customer: # read in the customer details from a CSV file

    def __init__(self, path):
        self.shopping_list = [] # define an empty shopping list (will append to later)
        with open(path) as csv_file:
            csv_reader = csv.reader(csv_file, delimiter=',')
            first_row = next(csv_reader)
            self.name = first_row[0] # treating the first row outside the for loop, defines the cust
            self.budget = float(first_row[1]) # treating the first row outside the for loop, defines
            for row in csv_reader: # for loop that loops through the CSV file and appends the produc
                name = row[0]
                quantity = float(row[1])
                p = Product(name)
                ps = ProductStock(p, quantity)
                self.shopping_list.append(ps)

    def calculate_costs(self, shop_stock): # function to define the cost of the products in the cust
        for i in shop_stock: # loops through the shop stock
            for j in self.shopping_list: # loops through the customer shopping list
                if (j.name() == i.name()): # checks for a match
                    j.product.price = i.unit_price() # where theres a match, set the price
```

Figure 3 Customer class (part of) in object oriented

With reference to Figure 2 and Figure 3, in the procedural implementation the Customer class contains the state only (name, budget and shopping list), whereas in the object-oriented implementation the Customer class contains state and functionality. The functionality reads in the customer details from a csv file, calculates the costs and outputs information.

Considering that Customer is a class, or template, there can be many instances of it. In the shop assignment there are 3 instances of customer. Each of the 3 customers have different names, budgets, and shopping lists but all 3 use the Customer class.

The approach from this point forward in this report is to look at each of the 5 options in the menu from Figure 1 and discuss the OOP and procedural implementations.

2.1 Menu Option 1 - Shop Balance

The assignment specification states, *"The shop CSV should hold the initial cash value for the shop"*. For both implementations "Option 1" in the menu displays the cash value. For this, the same file "stock.csv" is used.

In the procedural implementation there are 2 functions "stock_shop()" and "print_shop_balance()". In the OOP implementation there is one method call to "shop.shop_balance()". In the procedural implementation "Shop" is a separate class

that's used within the "stock_shop" function whereas within the OOP implementation the "Shop" class includes state and function.

2.2 Menu Option 2 - Product Stock Levels and Pricing

Selecting Option 2 from the menu returns the product stock levels and pricing for the shop. This was not required per the assignment specification but felt it was worthwhile as it shows what's available and how much it costs. In the procedural implementation there's a "print_shop()" function. This function combined with the stock_shop() is used. In the OOP implementation this is much easier as the "Shop" class is used where there's a "stock_levels_pricing(self)" method.

2.3 Menu Option 3 - Read in Customer Order

The assignment specification states, *"Read in customer orders from a CSV file.*

- *That file should include all the products they wish to buy and in what quantity.*
- *It should also include their name and their budget."*

To provide identical "user experience" as per the assignment specification when Option 3 is selected the user is asked to select a customer, (this is the case for both implementations). There are 3 customers; A, B or C (discussed previously).

When the user selects a customer that order is processed. During processing the products that the customer wants to buy and in what quantity are displayed. Also included in both implementations is the customer's name and their budget as per the assignment specification.

In the procedural this is complete by using two functions, "read_customer()" is used to read in the customer details and print_customer is used to print the details to the console. In the OOP to get the same output only print(customer) is required. This is because when print is called it calls __repr__() from the customer class. One extra feature that was included in the OOP is the cost to the customer. This lets the customer know how much the order would cost, and individual item costs. This was not required from the assignment specification but was a good addition.

2.4 Menu Option 4 - Process Order

The assignment specification states, *"The shop must be able to process the orders of the customer.*

- *Update the cash in the shop based on money received.*
- *It is important that the state of the shop be consistent.*
- *You should create customer test files (CSVs) which cannot be completed by the shop e.g. customer wants 400 loaves of bread but the shop only has 20, or the customer wants 2 cans of coke but can only afford 1.*
- *Know whether or not the shop can fill an order and throw an appropriate error"*

In both implementations the cash is updated based on the money received. This is complete by subtracting the purchase cost from the customer budget and adding the purchase cost to the shop cash. The state of the shop is consistent whereby if multiple orders are complete the shop cash balance accumulates.

There are three customer test files that are commonly used across both implementations, they are A.csv, B.csv and C.csv (detailed above) that satisfy the criteria per the assignment specification.

If the shop can't fulfil an order it'll throw an error per Figure 4.

```
There's 10 Bread in stock, your order of 1 can be satisied leaving 9 remaining in stock.
The unit cost of Bread is 0.7 so the total cost for 1 is 0.7.
Your budget is 94.7 leaving a balance of 94.0

*** WARNING *** Order can't be complete, there's insufficient stock levels for Spaghetti we have 5 in stock but you require 6.

*** WARNING *** Order can't be complete, there's insufficient stock levels for Jam we have 3 in stock but you require 8.

There's 10 Yoghurt in stock, your order of 4 can be satisied leaving 6 remaining in stock.
The unit cost of Yoghurt is 1.7 so the total cost for 4 is 6.8.
Your budget is 94.0 leaving a balance of 87.2

No Apple in stock
```

Figure 4 error insufficient stock levels(spaghetti, jam) and no stock levels (apples)

In the procedural implementation there are two functions used: stock_shop() (stocks the shop using the stock.csv file) and online_order() (processes the order). In the OOP implementation method passing using the customer object and the shop object are used.

The customer object calls the calculate_costs() method and the shop object calls the check_shop() and the order_processing() methods.

2.5 Menu Option 5 - Live Mode

The assignment specification states, *"Operate in a live mode, where the user can enter a product by name, specify a quantity, and pay for it. The user should be able to buy many products in this way."*

Both the procedural and OOP implementations operate in a live mode. When option 5 is selected the user is asked for 4 inputs:

1. Name
2. Budget
3. Item
4. Quantity

These inputs are then written to a csv file called "live.csv". This file is overwritten for every new live instance.

Writing to and using this csv file allows the same functions and objects to be used providing the same user experience as processing an order from the customer test csv files as per the project specification.

In the live mode of the OOP implementation the Live object calls the method write_to_csv which writes the customer and order details to the live.csv file. Then the functionality to process the order is the same as option 4.

The live mode of the procedural implementation is similar to the OOP, there is a function called write_to_csv that takes the user information, writes it to the live.csv file. Then the functionality to process the order is the same as option 4.

The user can continue to add items to the order until either "x" is selected as an item or "0" is selected as a quantity. If the shop can't fulfil an order it'll throw an error per Figure 4. This results in the same user experience in all modes and in all implementations.