Unit 11

Web Development in Python (Assignment)

Unit 11 System Implementation

The solution code represents the realisation of the UML diagram presented in Unit 7. Each UML class is represented by an associated Python source code file. For example, in the UML diagram, there exists a class titled "PaymentDetails". To locate the associated Python source code file, please find it using the exact same UML class name, "PaymentDetails.py"

Structure

All source files and tests files for the UML model, are found under the **src** folder.

Python Solution

To run the test files, please ensure Python version 3.9 is installed on the machine. Then, open each file separately in a relevant Python editor and run each script. Output is provided for Basic tests via `print()` statements, while process flow shows *how* the system meets the requirements specified in unit 7.

Tests

Although *pytest* is the recommended tool for testing Python code, this tool is not leveraged for testing the implementation due to:

1. Lack of familiarity with the tool.
2. Time constraints for this module that prevent building a real-world system that stands the test of time.

UML activity test file

The UML activity flow is found in the test file __**process**__**.py**. This file tests the model from the standpoint of a developer using the Python solution. The entire process from beginning to end is coded and shows how the system from unit 7 is translated into a similar Python project.

Each step in the UML activity diagram is identified in Python code with the code comment

```
# [UML_Activity_Step]: <step name>
...
```

Note: It was only while working with the process flow in actual code, that a few changes were identified that differed slightly form the UML model of unit 7.

UML Order's state test file

The process test file __**state.py**__ tests an order's state changes as it moves through system as described by the UML state model.

Other tests

The basic tests file __**tests.py**__ tests various concepts of the UML model, such as:

1. Inheritance.
2. Polymorphism.
3. Composition.

Assumptions

- It is best to keep implementations as simple as possible. Enhancements can always be made further down the line, but experience has taught me that it is better to demonstrate a *working concept* (also known as Minimum Viable Product, Moogk (2012)) rather than a complex, perfect system that will never be delivered.
- This assignment is neither a *report* nor an *essay*, therefore few (if any) academic references are referenced with regards to the design or implementation.

Design Considerations

- The implementation follows the design specification of the UML model developed in unit 7. Any additional functionality is kept to a minimum.
- Each UML class is represented by a separate Python code file.

- Complexities, such a validating credit card numbers, are excluded from this implementation.
- Searching for products is based on results from a single seller. Multiple products from multiple sellers are not supported in this implementation.
- There is no requirement for *multiple inheritance* (as dictated by the UML model). Therefore, contemplations surrounding **multiple resolution order** are not considered.
- No GUI framework was leveraged for this implementation.
- Concurrency concerns were considered, but not required for this basic implementation.
- All collections of data are represented as Python **lists** (not **sets**). Type hinting is used for some methods that return list data, for example

  ```
  def search_products(self, query) -> List[Product]:
     ...
  ```

  Here, `List[Product]` provides type hints that the return type is a list type that contains a collection of product instances

- **dictionaries** are not used extensively because their use case is not warranted in the implementation.
- **tuples** are not used because all data used in the implementation is of a *known* data structure.
- Naming conventions follow standard Python practice of using underscore (`_`) between words

```
def search_for_products():
    ...
```

Unit 7 Feedback

Feedback from unit 7 module assignment was considered,

"in the state activity diagram processing state could have been decomposed to include both awaiting delivery choice and awaiting payment choice" (Beran, 2021)

Upon reflection, I think the existing **OrderState** enumeration already reflects the requirements fro the model of unit 7, namely `AwaitingPayment` and `AwaitingPicking` of the `OrderState` enumeration. *Please see the test file __state.py__*

Updates since Unit 7

Save for the following additions listed below, no further changes were made to the Python source code files. This is keeping in line with the requirement to matching closely both the UML diagram and source code.

Changes introduced into the source code, include:

- **PaymentDetails.py**

  Make use of a callback to notify after a child instance of `PaymentDetails` has successfully processed a given payment.

```
class PaymentDetails:

    def __init__(self, account_number, sort_code,
payment_submitted_callback = None) -> None:
    ...

    def submit_payment(self, amount):

        result = self._handle_payment(amount)
        if result == True and self.__payment_submitted is not None:
            self.__payment_submitted()

        return result

    def _handle_payment(self, amount):
        pass
```

- **Seller.py**

  A `delivery_options()` method was added to facilitate determination of supported delivery options available for a particular `Seller` instance.

```
class Seller:
    ...
    def delivery_options(self) -> List[DeliveryOption]:
    ...
```

  Also, the source code introduced a method to initialise all product prices to their default values (when products are added to the seller's catalogue):

```
def _init_price_list(self):

    ...
```

- **OrderCheckout.py**

  Added a `calculate()` method to facilitate updating the total amount payable by a customer and also to incorporate any discounts from applied promo codes.

```
class OrderCheckout:
    ...
    def calculate(self):
        ...
```

- **Package.py**

  The UML diagram did not show how a collection of products would be added to the warehouse package.

```
class Package:
    ...
    def add_product(self, product: Product):
        ...
```

- **Product.py**

  A `weight` attribute was added to add the concept of pay-by-weight delivery. This attribute is more a nice-to-have, but one I can see a utilised in a real-world system.

```
class Product:
    def __init__(self, name, description, price, product_code):
        ...
        self.weight = random.randint(0, 200)
```

- **PromotionCode.py**

  A `discount_amount` attribute was added to make it obvious *how* a promotion code brings value to a real-world system. This attribute is used by the `calculate()` method introduced into the **OrderCheckout.py** source file.

```
class PromotionCode:
    def __init__(self) -> None:
        ...
        self.discount_amount = 0.25
```

Python Features Utilised

The solution code presented utilised the following features of Python:

1. Inheritance.
2. Polymorphism.
3. Private instance variables (prefixed with double underscore, (`__`).
4. Protected instance variables (prefixed with single underscore, (`_`).
5. Instance properties using the `@property` decorator.

6. Class variables. *See OrderCheckout.py*
7. Type hinting where such hints add clarity to method definitions:

```
def xxx() -> <type>
```

8. Nested methods.

```
def main_outer_method():
        def inner_method():
            ...
        ...
```

9. **hasattr()**, **issubclass()** and **repr()** function. `__repr()__` is used to make friendly representations of classes during debug.
10. Definition of an enum class using the Python **enum** module. *See OrderState.py*.
11. Local `import` statements inside methods -- useful to resolve circular dependencies. *See WarehouseStaff::package_next_order()*

Resources

Beran (2021) Module submission feedback. Available from https://www.my-course.co.uk/mod/assign/view.php?id=496414 [Accessed on 15 Jul 2021]

Moogk, D.R. (2012) Minimum viable product and the importance of experimentation in technology startups. *Technology Innovation Management Review*, 2(3).