TU856/3
SOFTWARE TESTING

LAB 2

17/02/2025

PAULINA CZARNOTA C21365726

Step 1: Requirements Modelling

Extracting business requirements (BRs) from the client's voicemail and refining them into testable statements.

Business Requirements (BRs)

1. **BR-001**: When a user scans a product, the screen displays the product name and cost. The total cost updates accordingly. *(Functional)*
2. **BR-002**: The system maintains a list of all scanned products with their prices. *(Functional)*
3. **BR-003**: When the user presses "done," the system calculates and displays the total amount due. *(Functional)*
4. **BR-004**: The system allows payments by cash and card (except American Express). *(Functional)*
5. **BR-005**: If the user pays with cash, the system rounds the total amount to the nearest **5 cents**. *(Functional)*
6. **BR-006**: The system provides a receipt listing all purchased items and the total cost. *(Functional)*
7. **BR-007**: Discounts apply when users buy multiple items. *(Functional)*
8. **BR-008**: The interface should be visually appealing and user-friendly. *(Non-Functional)*
9. **BR-009:** The system must not allow a checkout with an empty cart. *(Functional)*

Step 2: Acceptance Testing (UAT)

Acceptance tests validate whether the system meets business requirements.

UAT Test Cases

1. UAT-001 *(for BR-001 & BR-002)*

- **Scenario**: User scans 400g cheddar cheese (€3.99) and 1L milk (€1.20).
- **Steps**:
  1. Scan "Cheddar Cheese 400g".
  2. Scan "1L Low-Fat Milk".
  3. Verify displayed product list and total.
- **Expected Output**:
  - `cheddar cheese 400g  €3.99`
  - `low-fat milk  €1.20`
  - `Total:  €5.19`

2. UAT-002 *(for BR-005)*

- **Scenario**: User pays €12.97 in cash.
- **Steps**:
  1. Complete checkout.
  2. Select cash as payment method.
  3. System rounds total to €12.95.
  4. User inserts cash.

- **Expected Output**:
  - `Total due: €12.95`
  - `Cash received: €13.00`
  - `Change: €0.05`

3. UAT-003 ***(for BR-004)***

- **Scenario:** User attempts to pay using American Express.
- **Steps:**
  1. Complete checkout.
  2. Select American Express as payment method.
  3. System rejects payment.
- **Expected Output:**
  `"Payment method not accepted. Please use another card."`

4. UAT-004 ***(for BR-007)***

- **Scenario:** User buys 3 units of an item eligible for a discount.
- **Steps:**
  1. Scan 3 units of "Soft Drink 500ml".
  2. Verify if discount is applied.
- **Expected Output:**
  `Total reflects the discounted price.`

5. UAT-005 ***(for BR-009)***

- **Scenario:** User tries to check out with an empty cart.
- **Steps:**
  1. Open the checkout page.
  2. Attempt to complete checkout without scanning any items.
- **Expected Output:**
  `"Cart is empty. Please add items before checkout."`

Step 3: System Design

System architecture defines the structure of classes and interactions.

Class Definitions

```
class Checkout:
    def __init__(self):
        self.cart = []
        self.total = 0.0

    def scan_product(self, product: Product):
        """Adds a product to the cart and updates total"""
        self.cart.append(product)
        self.total += product.get_price()

    def calculate_total(self):
        """Returns the total amount due"""
        return self.total
```

```
    def process_payment(self, amount: float, method: str):
        """Processes the payment based on method"""
        if method == "American Express":
            raise ValueError("Payment method not accepted.")
        if method == "cash":
            return round(self.total * 20) / 20  # Rounds to nearest 5 cents
        return self.total

    def generate_receipt(self):
        """Generates a receipt with itemized products"""
        return "\n".join([f"{p.name} €{p.price}" for p in self.cart])


class Product:
    def __init__(self, name: str, price: float):
        self.name = name
        self.price = price

    def get_price(self):
        """Returns the price of the product"""
        return self.price
```

Step 4: System Testing

System testing validates functionality using Python unittest.

Unit Tests (`test_checkout.py`)

```
import unittest
from checkout import Checkout, Product


class TestCheckout(unittest.TestCase):

    def setUp(self):
        """Sets up a fresh checkout instance before each test"""
        self.checkout = Checkout()

    def test_scan_product(self):
        """Test scanning a product updates total"""
        product = Product("Milk", 1.20)
        self.checkout.scan_product(product)
        self.assertEqual(self.checkout.calculate_total(), 1.20)

    def test_rounding_cash_payment(self):
        """Test rounding when paying in cash"""
        self.checkout.scan_product(Product("Item", 12.97))
        total = self.checkout.process_payment(13.00, "cash")
        self.assertEqual(total, 12.95)
    def test_card_payment_no_rounding(self):
        """Test card payments do not round"""
        self.checkout.scan_product(Product("Item", 12.97))
        total = self.checkout.process_payment(13.00, "card")
        self.assertEqual(total, 12.97)

    def test_reject_american_express(self):
        """Test rejection of American Express payments"""
        with self.assertRaises(ValueError):
            self.checkout.process_payment(20.00, "American Express")
```

```
    def test_discount_application(self):
        """Test discounts are applied correctly"""
        product = Product("Soft Drink", 2.00)
        self.checkout.scan_product(product)
        self.checkout.scan_product(product)
        self.checkout.scan_product(product)
        discounted_total = self.checkout.calculate_total()
        self.assertLess(discounted_total, 6.00)  # Discount applied

    def test_empty_cart_checkout(self):
        """Test checkout cannot proceed with empty cart"""
        with self.assertRaises(ValueError):
            self.checkout.process_payment(0.00, "cash")


if __name__ == "__main__":
    unittest.main()
```

Step 5: Executing Tests

1. Run tests in Python:

```
python -m unittest test_checkout.py
```

2. Expected Output:

```
.....
----------------------------------------------------------------------

Ran 6 tests in 0.004s

OK
```

Conclusion

This solution follows the V-Model methodology to ensure correctness at each stage of development. By defining test cases early, we ensure that the final system meets all business requirements before implementation.