# shopping\_cart.py

class Product:

def \_\_init\_\_(self, product\_id, name, price):

self.product\_id = product\_id

self.name = name

self.price = price

def \_\_repr\_\_(self):

return f"Product({self.product\_id}, {self.name}, {self.price})"

class Cart:

def \_\_init\_\_(self):

# Dictionary mapping product\_id to a tuple (Product, quantity)

self.items = {}

def add\_product(self, product, quantity=1):

if quantity <= 0:

raise ValueError("Quantity must be positive")

if product.product\_id in self.items:

current\_qty = self.items[product.product\_id][1]

self.items[product.product\_id] = (product, current\_qty + quantity)

else:

self.items[product.product\_id] = (product, quantity)

def remove\_product(self, product, quantity=1):

if product.product\_id not in self.items:

raise ValueError("Product not in cart")

if quantity <= 0:

raise ValueError("Quantity must be positive")

current\_qty = self.items[product.product\_id][1]

if quantity >= current\_qty:

del self.items[product.product\_id]

else:

self.items[product.product\_id] = (product, current\_qty - quantity)

def calculate\_total(self):

total = 0.0

for product, quantity in self.items.values():

total += product.price \* quantity

return total

def apply\_discount(self, discount\_rate):

if discount\_rate < 0 or discount\_rate > 100:

raise ValueError("Discount rate must be between 0 and 100")

total = self.calculate\_total()

discount\_amount = total \* (discount\_rate / 100.0)

return total - discount\_amount

class Order:

def \_\_init\_\_(self, cart, customer\_name):

self.cart = cart

self.customer\_name = customer\_name

self.total\_amount = cart.calculate\_total()

self.status = "Pending"

def process\_order(self):

if self.total\_amount <= 0:

raise ValueError("Cannot process order with zero total")

self.status = "Processed"

return True

class Inventory:

def \_\_init\_\_(self):

# Dictionary mapping product\_id to available quantity

self.stock = {}

def add\_stock(self, product, quantity):

if quantity <= 0:

raise ValueError("Quantity must be positive")

if product.product\_id in self.stock:

self.stock[product.product\_id] += quantity

else:

self.stock[product.product\_id] = quantity

def remove\_stock(self, product, quantity):

if product.product\_id not in self.stock or self.stock[product.product\_id] < quantity:

raise ValueError("Insufficient stock")

self.stock[product.product\_id] -= quantity

def check\_stock(self, product):

return self.stock.get(product.product\_id, 0)

class Coupon:

def \_\_init\_\_(self, code, discount\_rate):

if discount\_rate < 0 or discount\_rate > 100:

raise ValueError("Invalid discount rate")

self.code = code

self.discount\_rate = discount\_rate

def apply\_coupon(self, cart):

return cart.apply\_discount(self.discount\_rate)

Test Cases:

import unittest

from shopping\_cart import Product, Cart, Order

# ==============================

# Test A

# ==============================

class TestAShoppingCart(unittest.TestCase):

def test\_end\_to\_end\_order\_process(self):

# Create actual objects from the shopping\_cart system

cart = Cart()

product1 = Product(1, "Widget", 10.0)

product2 = Product(2, "Gadget", 20.0)

# Simulate user actions: adding products to the cart.

cart.add\_product(product1, 2) # Total: 2 \* 10.0 = 20.0

cart.add\_product(product2, 1) # Total: 1 \* 20.0 = 20.0

# Combined total should be 40.0

# Create an order using the actual cart

order = Order(cart, "Alice")

result = order.process\_order()

# Verify the order processed successfully and the total amount is correct.

self.assertTrue(result)

self.assertEqual(order.status, "Processed")

self.assertAlmostEqual(order.total\_amount, 40.0)

# ==============================

# Test B

# ==============================

class StubCart:

def calculate\_total(self):

# Return a fixed total for testing purposes.

return 100.0

# If Order were to invoke other Cart methods, add stubs here as needed.

class TestB (unittest.TestCase):

def test\_\_with\_stub\_cart(self):

# Use the stub in place of the real Cart.

stub\_cart = StubCart()

order = Order(stub\_cart, "Bob")

result = order.process\_order()

# Validate that the order is processed correctly with the stub value.

self.assertTrue(result)

self.assertEqual(order.status, "Processed")

self.assertAlmostEqual(order.total\_amount, 100.0)

# ==============================

# Run Tests

# ==============================

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()