# 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, Inventory, Coupon

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

# TestA

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

class TestProduct(unittest.TestCase):

def test\_product\_creation(self):

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

self.assertEqual(product.product\_id, 1)

self.assertEqual(product.name, "Widget")

self.assertEqual(product.price, 10.0)

class TestCart(unittest.TestCase):

def setUp(self):

self.cart = Cart()

self.product = Product(1, "Widget", 10.0)

def test\_add\_and\_calculate\_total(self):

self.cart.add\_product(self.product, 3)

total = self.cart.calculate\_total()

self.assertEqual(total, 30.0)

def test\_remove\_product(self):

self.cart.add\_product(self.product, 5)

self.cart.remove\_product(self.product, 2)

# Remaining quantity should be 3

self.assertEqual(self.cart.items[self.product.product\_id][1], 3)

def test\_apply\_discount(self):

self.cart.add\_product(self.product, 4) # Total = 40.0

discounted\_total = self.cart.apply\_discount(25) # 25% discount

self.assertAlmostEqual(discounted\_total, 30.0)

class TestOrder(unittest.TestCase):

def setUp(self):

self.cart = Cart()

self.product = Product(1, "Widget", 15.0)

self.cart.add\_product(self.product, 2) # Total = 30.0

self.order = Order(self.cart, "Alice")

def test\_order\_total(self):

self.assertAlmostEqual(self.order.total\_amount, 30.0)

def test\_process\_order(self):

self.assertTrue(self.order.process\_order())

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

class TestInventory(unittest.TestCase):

def setUp(self):

self.inventory = Inventory()

self.product = Product(1, "Widget", 10.0)

def test\_add\_stock(self):

self.inventory.add\_stock(self.product, 50)

self.assertEqual(self.inventory.check\_stock(self.product), 50)

def test\_remove\_stock(self):

self.inventory.add\_stock(self.product, 50)

self.inventory.remove\_stock(self.product, 20)

self.assertEqual(self.inventory.check\_stock(self.product), 30)

class TestCoupon(unittest.TestCase):

def setUp(self):

self.cart = Cart()

self.product = Product(1, "Widget", 100.0)

self.cart.add\_product(self.product, 1) # Total = 100.0

def test\_apply\_coupon(self):

coupon = Coupon("SAVE10", 10)

discounted\_total = coupon.apply\_coupon(self.cart)

self.assertAlmostEqual(discounted\_total, 90.0)

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

# Test B

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

class TestShoppingCart(unittest.TestCase):

def test\_cart\_stability\_under\_repeated\_operations(self):

cart = Cart()

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

iterations = 10000 # Simulate repeated operations

for \_ in range(iterations):

cart.add\_product(product, 1)

cart.remove\_product(product, 1)

# After many iterations, the cart total should be 0.

self.assertEqual(cart.calculate\_total(), 0.0)

def test\_order\_under\_repeated\_processing(self):

product = Product(1, "Widget", 20.0)

iterations = 1000 # Number of orders processed consecutively

for i in range(iterations):

cart = Cart()

cart.add\_product(product, 5) # Total = 5 \* 20.0 = 100.0

order = Order(cart, f"Customer\_{i}")

order.process\_order()

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

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

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

unittest.main()