# 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:

# test\_shopping\_cart.py

import SomeTypeOfTest

from shopping\_cart import Product, Cart, Order, Inventory, Coupon

//A

class TestCartOperations(SomeTypeOfTest.Test):

def setUp(self):

self.prod1 = Product(1, "Widget", 10.0)

self.prod2 = Product(2, "Gadget", 20.0)

self.cart = Cart()

def test\_add\_product(self):

self.cart.add\_product(self.prod1, 2)

self.assertEqual(self.cart.items[self.prod1.product\_id][1], 2)

self.cart.add\_product(self.prod1, 3)

self.assertEqual(self.cart.items[self.prod1.product\_id][1], 5)

def test\_remove\_product(self):

self.cart.add\_product(self.prod1, 3)

self.cart.remove\_product(self.prod1, 1)

self.assertEqual(self.cart.items[self.prod1.product\_id][1], 2)

self.cart.remove\_product(self.prod1, 2)

self.assertNotIn(self.prod1.product\_id, self.cart.items)

//B

def test\_remove\_nonexistent\_product\_ (self):

with self.assertRaises(ValueError):

self.cart.remove\_product(self.prod1, 1)

def test\_calculate\_total(self):

self.cart.add\_product(self.prod1, 1)

self.cart.add\_product(self.prod2, 2)

total = self.cart.calculate\_total()

self.assertAlmostEqual(total, 10.0 + 2 \* 20.0)

def test\_apply\_discount(self):

self.cart.add\_product(self.prod1, 2) # Total = 20.0

discounted\_total = self.cart.apply\_discount(10)

self.assertAlmostEqual(discounted\_total, 20.0 \* 0.9)

def test\_invalid\_discount(self):

with self.assertRaises(ValueError):

self.cart.apply\_discount(150)

def test\_invalid\_quantity\_add(self):

with self.assertRaises(ValueError):

self.cart.add\_product(self.prod1, 0)

def test\_invalid\_quantity\_remove(self):

self.cart.add\_product(self.prod1, 2)

with self.assertRaises(ValueError):

self.cart.remove\_product(self.prod1, 0)

class TestOrderProcessing(SomeTypeOfTest):

def setUp(self):

self.prod1 = Product(1, "Widget", 10.0)

self.cart = Cart()

self.cart.add\_product(self.prod1, 3) # 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\_success(self):

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

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

def test\_process\_empty\_order(self):

empty\_cart = Cart()

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

with self.assertRaises(ValueError):

order.process\_order()

class TestInventoryOperations(SomeTypeOfTest):

def setUp(self):

self.inventory = Inventory()

self.prod1 = Product(1, "Widget", 10.0)

def test\_add\_and\_check\_stock(self):

self.inventory.add\_stock(self.prod1, 50)

self.assertEqual(self.inventory.check\_stock(self.prod1), 50)

def test\_remove\_stock(self):

self.inventory.add\_stock(self.prod1, 30)

self.inventory.remove\_stock(self.prod1, 10)

self.assertEqual(self.inventory.check\_stock(self.prod1), 20)

with self.assertRaises(ValueError):

self.inventory.remove\_stock(self.prod1, 25)

class TestCouponFunctionality(SomeTypeOfTest):

def setUp(self):

self.cart = Cart()

self.prod1 = Product(1, "Widget", 100.0)

self.cart.add\_product(self.prod1, 1)

self.coupon = Coupon("DISCOUNT10", 10)

def test\_apply\_coupon(self):

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

self.assertAlmostEqual(discounted\_total, 90.0)

def test\_invalid\_coupon\_discount(self):

with self.assertRaises(ValueError):

Coupon("BADCOUPON", 150)

# Running all tests

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

unittest.main()