Software Testing CW Report

Student Name: Fahad Alhssan - K25049365

Course: 6CCS3SMT Software Measurement and Testing

Testing Approach

I used a requirements-based testing approach, creating test cases directly from the system requirements to ensure all specified functionality is validated.

For the more complex requirements, I applied **equivalence partitioning** to systematically identify different input scenarios. For example, in Requirement 3 (bundle discount), I identified different classes based on the number of laptops and mice in the cart: cases with no laptops, no mice, equal quantities, more mice than laptops, and more laptops than mice. This technique helped me create a comprehensive set of test cases that cover all logical scenarios without redundant tests.

I organized tests by requirement, creating separate test files for different features:

- `test_browsing_and_adding.py` - Basic cart functionality (Requirements 1-2)
- `test_bundle_discount.py` - Bundle discount logic (Requirement 3)
- and so on.

This organization makes it easier to maintain tests and understand which requirements are being validated.

for the framework, I chose **pytest** for Python testing because it provides clear output, simple assertions, and good test organization capabilities.

2. Test Cases and Coverage

I will only discuss requirements that had failing tests. to maintain the word limits and make the report concise.

Requirement 3: Bundle Discount (Mouse-Laptop Pairs)

"The system shall apply a bundle discount: 10% off the price of each mouse if at least one laptop is in the cart. This discount applies for all mouse-laptop pairs."

This requirement was more complex to understand. After clarification from the Q&A forum, I learned that each laptop can discount only one mouse, following a min(laptops, mice) pairing system.

Test Cases Designed:

I created 8 test cases to systematically verify the bundle discount logic and cover each possible logical combinations of input:

1. `test_no_bundle_discount_without_laptop` - No laptop in cart, no discount should apply (PASSED)
2. `test_no_bundle_discount_without_mouse` - No mouse in cart, no discount should apply (FAILED)
3. `test_one_laptop_one_mouse` - 1:1 pairing, 1 mouse should get 10% off (FAILED)
4. `test_one_laptop_multiple_mice` - 1 laptop + 3 mice, only 1 mouse should get discount (FAILED)
5. `test_two_laptops_two_mice` - Perfect 2:2 pairing, both mice should get discount (FAILED)
6. `test_two_laptops_three_mice` - 2 laptops + 3 mice, only 2 mice should get discount (FAILED)
7. `test_three_laptops_two_mice` - More laptops than mice, both mice should get discount (FAILED)
8. `test_bundle_discount_with_zero_price_mouse` - Edge case with free mouse (FAILED)

The tests used carefully chosen prices (£200 laptops, £150 laptops for tests 5-7, and £100 mice) to avoid accidentally triggering other discounts and to clearly distinguish between correct behavior and buggy behavior.

Detected Faults:

Class Name	Line Number(s)	Description of Fault	Test Case(s) That Expose the Fault
Discount Service	Line 18	**Inverted condition**, The code checks `if item != "mouse"` instead of `if item == "mouse"`. This causes the 10% bundle discount to be applied to laptops (and any other non-mouse products) instead of to mice, completely inverting the intended behavior.	`test_no_bundle_discount_without_mouse`, `test_one_laptop_one_mouse`, `test_one_laptop_multiple_mice`, `test_two_laptops_two_mice`, `test_two_laptops_three_mice`, `test_three_laptops_two_mice`, `test_bundle_discount_with_zero_price_mouse`
Discount Service	Line 23	**Missing quantity handling**, The code uses `item.get_product().get_price() * 0.10` without multiplying by `item.get_quantity()`. When a CartItem has quantity > 1, only one unit's price gets discounted instead of all units that should be discounted based on the pairing logic. This means the loop processes CartItems rather than individual product units.	`test_two_laptops_two_mice`, `test_two_laptops_three_mice`, `test_three_laptops_two_mice`

These two bugs compound each other, not only is the wrong product type being discounted, but only one unit is discounted regardless of how many pairs should exist according to the requirement.

Requirement 4: Tiered Discounts

"The system shall apply tiered discounts based on the cart subtotal after bundle discounts:
• 15% off if the subtotal exceeds £2,000
• 20% off if the subtotal exceeds £7,000
• 25% off if the subtotal exceeds £15,000"

This requirement introduces tiered discounts that should be applied after bundle discounts. The system should select the highest applicable tier based on the subtotal, and the discount should only apply when the subtotal strictly exceeds (is greater than) the threshold.

Test Cases Designed:

I created 8 test cases to systematically verify the tiered discount logic with boundary testing and scenarios combining bundle discounts:

1. `test_no_tiered_discount_below_threshold` - £500 subtotal, well below any threshold (PASSED)
2. `test_no_discount_at_1999` - £1999 subtotal, just below £2000 threshold, should get no discount (FAILED)
3. `test_apply_15_without_bundle` - £2001 subtotal exceeds £2000, should get 15% off (PASSED)
4. `test_apply_15_with_bundle` - Bundle discount brings subtotal above £2000 (FAILED)
5. `test_apply_20_without_bundle` - £7001 subtotal exceeds £7000, should get 20% off (PASSED)
6. `test_apply_20_with_bundle` - Bundle discount brings subtotal above £7000 (FAILED)
7. `test_apply_25_without_bundle` - £15001 subtotal exceeds £15000, should get 25% off (PASSED)
8. `test_apply_25_with_bundle` - Bundle discount brings subtotal above £15000 (FAILED)

Detected Faults:

Class Name	Line Number(s)	Description of Fault	Test Case(s) That Expose the Fault
DiscountService	Line 30	**Incorrect threshold value**, Uses `total > 1000` instead of `total > 2000` for the 15% discount tier. This causes the 15% discount to be incorrectly applied to any subtotal above £1000, affecting the £1001-£2000 price range.	`test_no_discount_at_1999`
DiscountService	Lines 17-23	**Bundle discount bug** (from Requirement 3), Applies discount to laptops instead of mice, causing incorrect subtotals to be passed to tiered discount logic. This cascades into all tests that combine bundle and tiered discounts.	All`*_with_bundle` test cases

Requirement 5: Customer Categories

"The system shall categorize customers into three types: Regular, Premium, and VIP, with Premium customers receiving an additional 10% discount and VIP customers receiving an additional 15% discount on their total."

This requirement introduces customer-specific discounts that should be applied based on customer type. To properly isolate and test this functionality, I focused exclusively on customer category discounts without combining them with other discount types (bundle or tiered).

Test Cases Designed:

I created 6 test cases to verify customer category discounts work correctly in isolation:

1. `test_regular_customer_no_additional_discount` - Single item, no discount (PASSED)
2. `test_premium_customer_gets_10_percent_discount` - Single item, 10% discount (FAILED)
3. `test_vip_customer_gets_15_percent_discount` - Single item, 15% discount (PASSED)
4. `test_regular_customer_with_multiple_items` - Multiple items, no discount (PASSED)
5. `test_premium_customer_with_multiple_items` - Multiple items, 10% discount (FAILED)
6. `test_vip_customer_with_multiple_items` - Multiple items, 15% discount (PASSED)

Detected Faults:

Class Name	**Line Number(s)**	**Description of Fault**	**Test Case(s) That Expose the Fault**
DiscountService	Line 35	**Incorrect discount percentage**, Uses `discount += 0.20` (20%) instead of `discount += 0.10` (10%) for Premium customers. This gives Premium customers double the intended discount, contradicting the requirement specification.	`test_premium_customer_gets_10_percent_discount`, `test_premium_customer_with_multiple_items`

Requirement 7: Time-Limited Promotions & Requirement 8: Discount Order

Requirement 7: "The system shall support time-limited promotions that can be activated or deactivated. During an active promotion, a flat discount of 25% shall be applied. This discount shall be applied in addition to any other applicable discounts."

Requirement 8: "Discounts shall be applied in the following order. First, bundle discounts and fixed-amount coupon discounts are applied, reducing the overall total of the order. Next, all percentage-based discounts (including tiered cart value discounts, customer type discounts, percentage-based coupon codes, and time-limited promotions) are applied to the updated total."

These two requirements are closely related and tested together. Requirement 7 introduces promotional periods with 25% off, while Requirement 8 defines the precise order in which all discount types should be applied.

Test Cases Designed:

I created 9 test cases to verify promotion activation/deactivation and discount ordering:

1. `test_no_promotion_active` - No promotion, no discount (PASSED)
2. `test_promotion_active` - Promotion gives 25% off (PASSED)
3. `test_promotion_with_multiple_items` - Promotion works with multiple items (PASSED)
4. `test_promotion_combines_with_tiered_discount` - Promotion (25%) combines with tiered (15%) = 40% total (FAILED)
5. `test_promotion_toggle_off` - Promotion can be deactivated (PASSED)
6. `test_discount_order_bundle_then_percentage` - Bundle first, then promotion percentage (FAILED)
7. `test_discount_order_fixed_coupon_then_percentage` - Fixed coupon first, then promotion (FAILED)
8. `test_discount_order_percentage_coupon_with_promotion` - Percentage coupon combines with promotion (FAILED)
9. `test_discount_order_all_types_combined` - All discount types in correct order (FAILED)

Detected Faults:

Class Name	**Line Number(s)**	**Description of Fault**	**Test Case(s) That Expose the Fault**
ShoppingCart	Lines 42-46	**Incorrect promotion logic**, When promotion is active, the code uses an if/else structure that ONLY applies the 25% promotion discount via `apply_promotion_discount()` and completely skips all other discounts (bundle, tiered, customer type, coupons). This violates both requirements which state promotions should combine with other discounts.	`test_promotion_combines_with_tiered_discount`, `test_discount_order_bundle_then_percentage`, `test_discount_order_fixed_coupon_then_percentage`, `test_discount_order_percentage_coupon_with_promotion`, `test_discount_order_all_types_combined

3.9 Requirement 10: Payment Validation

"The system shall display an error message if the credit card number does not meet the 16-digit requirement or if the transaction amount is zero or negative, preventing the transaction from proceeding."

This requirement specifies validation rules for payment processing to prevent invalid transactions.

Test Cases Designed:

I created 10 test cases to verify payment validation:

1. `test_valid_payment` - Valid 16-digit card and positive amount (PASSED)
2. `test_card_less_than_16_digits` - 15-digit card should be rejected (FAILED)
3. `test_card_more_than_16_digits` - 17-digit card should be rejected (FAILED)
4. `test_zero_amount` - Zero amount should be rejected (FAILED)
5. `test_negative_amount` - Negative amount should be rejected (FAILED)
6. `test_invalid_card_and_zero_amount` - Both invalid card and zero amount (PASSED)
7. `test_invalid_card_and_negative_amount` - Both invalid card and negative amount (PASSED)
8. `test_empty_card_number` - Empty card number should be rejected (FAILED)
9. `test_very_small_positive_amount` - £0.01 should be accepted (PASSED)
10. `test_large_amount` - Large amount should be accepted (PASSED)

Detected Faults:

Class Name	Line Number(s)	Description of Fault	Test Case(s) That Expose the Fault
PaymentService	Line 5	**Incorrect logical operator**, Uses `and` instead of `or` in the validation condition. The code checks `if len(credit_card_number) != 16 and amount <= 0`, meaning it only raises an exception when both conditions are true. The requirement states the error should occur if either condition is true (invalid card OR invalid amount).	`test_card_less_than_16_digits`, `test_card_more_than_16_digits`, `test_zero_amount`, `test_negative_amount`, `test_empty_card_number`

4. Testing Challenges and Trade-offs

Initially, I misunderstood the requirement "applies for all mouse-laptop pairs." I thought all mice would get discounts if at least one laptop was present. After clarification, I learned it's a min(laptops, mice) pairing. This required me to redesign several test cases.

And in discounts test cases, the system had multiple discount types (bundle, tier, customer type, promotions) that interact with each other. I had to carefully calculate expected values by understanding the order in which discounts are applied.

I even thought there was a bug in some requirements from discount calculation, but this was due to another discount being triggered.

One trade-off was made is preferring Test Readability over DRY Principle. I repeated setup code in each test rather than reusing or refactoring the set up. This makes tests more verbose but easier to understand and maintain.

5. Code Coverage

I used **pytest-cov** (based on coverage.py) to measure code coverage. This tool tracks which lines of code are executed during test runs.

Output:

File	**Statements**	**Missed**	**Coverage**
CartItem.py	10	1	90%
Customer.py	15	4	73%
CustomerType.py	5	0	100%
DiscountService.py	31	0	100%
PaymentService.py	5	0	100%
Product.py	17	4	76%
ShoppingCart.py	37	1	97%
InventoryService.py	7	7	0%
OrderService.py	17	17	0%
TOTAL	**724**	**39**	***95%**

6. Mocking Strategy

I did not need to use any mocking, mocking will be helpful if we needed to test, for example, different payment scenarios. but the requirements and project were only focused on input validation. if we had to test how the system would react to payment failures due to insufficient funds, network errors and such, mocking would be useful, otherwise keeping it simple was the approach i went for.