TEST ANALYSIS DOCUMENTATION

Project: POS System
Team Member: Abdulaziz Bezan

---

1. EXECUTIVE SUMMARY

Analysis of three methods from CartUI.java for BVT, Decision Table, and Code Coverage testing.

Methods: calculateTotal(), addToCartBtnActionPerformed()

---

2. METHOD 1: calculateTotal()

```
private void calculateTotal() {
    double total = 0.0;
    for (int i = 0; i < cartTable.getRowCount(); i++) {
        // Extract price and quantity of each item
        double totalPrice = Double.parseDouble(cartTable.getValueAt(i, 3).toString());
        total += totalPrice;
    }
    totalofcart.setText(String.valueOf(total));
}
```

Purpose: Sums up cart for totel

2.1 Boundary Value Testing

Test Case ID	Input (cartTable state)	Expected Output	Test Type	Reason
BVT-CALC-1	0 rows (empty table)	totalofcart = "0.0"	Boundary	Minimum row count
BVT-CALC-2	1 row, value="10.00"	totalofcart = "10.00"	Boundary	Single item calculation

Test Case ID	Input (cartTable state)	Expected Output	Test Type	Reason
BVT-CALC-3	1 row, value="0.01"	totalofcart = "0.01"	Boundary	Minimum price value
BVT-CALC-4	1 row, value="999999.99"	totalofcart = "999999.99"	Robustness	Maximum practical price
BVT-CALC-5	2 rows: "100.50" + "200.75"	totalofcart = "301.25"	Typical	Multiple item sum
BVT-CALC-6	Multiple rows with decimals	Correct sum with 2 decimal places	Accuracy	Decimal precision
BVT-CALC-7	Value with 3+ decimals: "10.555"	Rounded/truncated appropriately	Robustness	Decimal formatting

- **This BVT tests the implicit boundaries of calculateTotal() by checking cart states (empty, single item, multiple items) and price values (minimum, maximum, decimals). The method works for basic cases, but lacks clear input parameters with defined ranges, making proper boundary testing difficult. Improvements would include adding explicit parameters with validation for better testability!**

2.2 Decision Table Testing

Condition	Rule 1	Rule 2	Rule 3
C1: Cart has items?	Yes	Yes	No

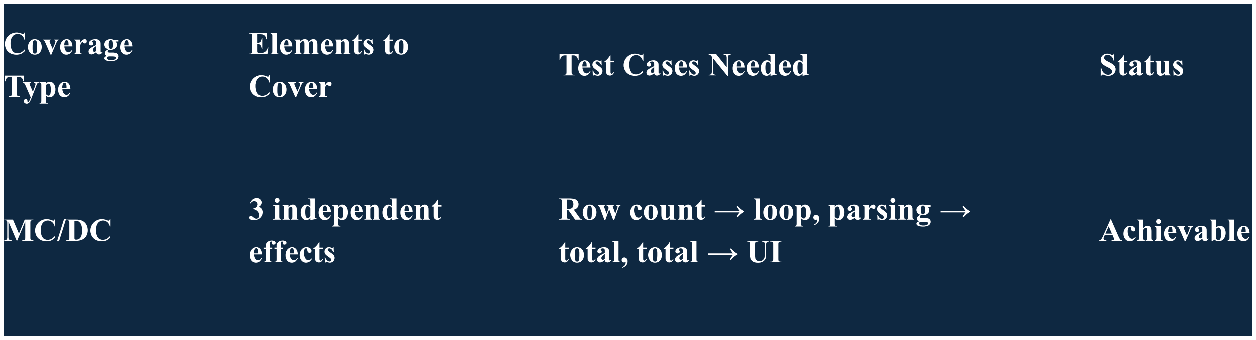
Condition	Rule 1	Rule 2	Rule 3
C2: All values numeric?	Yes	No	N/A
Action: Calculate sum	✓		
Action: Handle error		✓	
Action: Display 0.0			✓

- This decision table shows the method's logic paths: normal calculation when cart has numeric items, error handling for non-numeric data, and zero display for empty carts. The table reveals missing error recovery, non numeric values cause crashes instead of graceful handling. The logic is simple, but brittle without input validation.

  (note, the final submission will provide the input user test values)

2.3 Code Coverage Analysis

Coverage Type	Elements to Cover	Test Cases Needed	Status
Statement	6 code statements	Empty cart, 1 item, multiple items	Partial
Branch	4 decision points	Loop (0/1+ times), parse (success/fail)	Incomplete
Condition	4 boolean conditions	rowCount >0 (T/F), parsing (T/F)	Partial

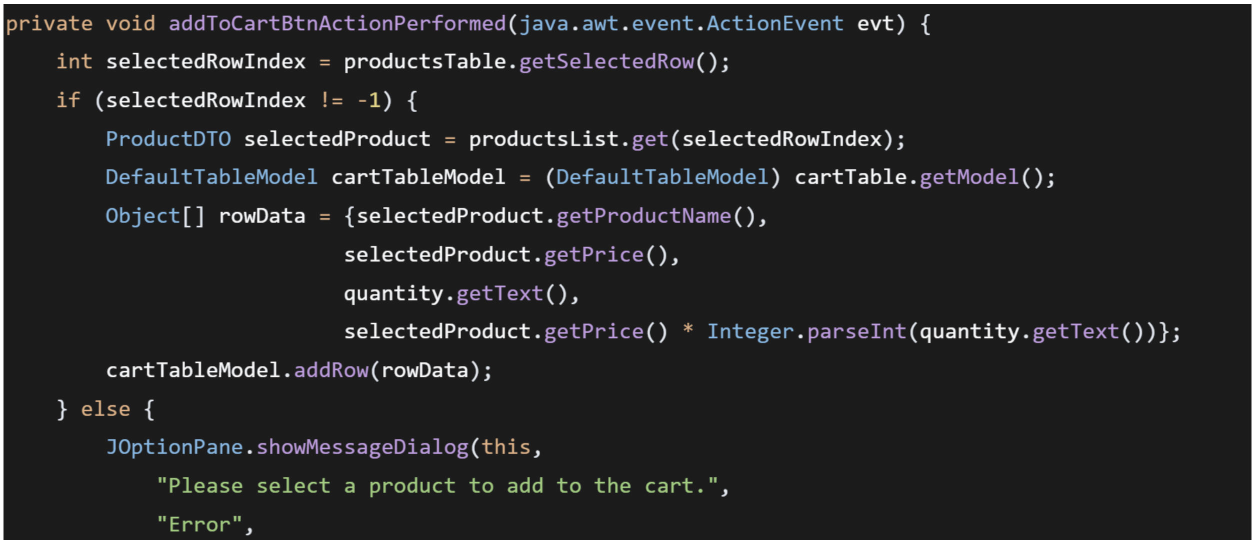


Coverage Type	Elements to Cover	Test Cases Needed	Status
MC/DC	3 independent effects	Row count → loop, parsing → total, total → UI	Achievable

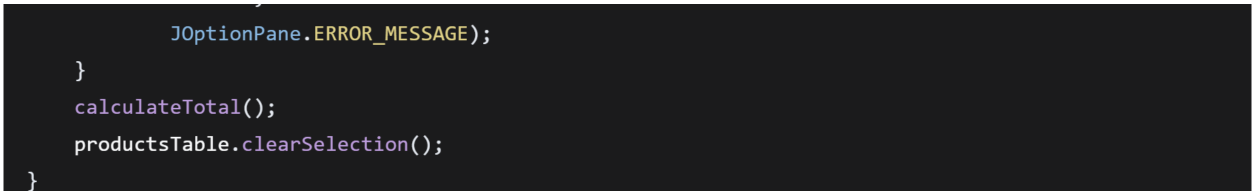
- This table shows calculateTotal() needs tests for all loop and parsing scenarios. Missing error handling prevents full branch coverage. MC/DC is achievable by testing each condition's independent effect on the output.

---

3. METHOD 2: addToCartBtnActionPerformed()



```
private void addToCartBtnActionPerformed(java.awt.event.ActionEvent evt) {
    int selectedRowIndex = productsTable.getSelectedRow();
    if (selectedRowIndex != -1) {
        ProductDTO selectedProduct = productsList.get(selectedRowIndex);
        DefaultTableModel cartTableModel = (DefaultTableModel) cartTable.getModel();
        Object[] rowData = {selectedProduct.getProductName(),
                            selectedProduct.getPrice(),
                            quantity.getText(),
                            selectedProduct.getPrice() * Integer.parseInt(quantity.getText())};
        cartTableModel.addRow(rowData);
    } else {
        JOptionPane.showMessageDialog(this,
            "Please select a product to add to the cart.",
            "Error",
```



```
            JOptionPane.ERROR_MESSAGE);
    }
    calculateTotal();
    productsTable.clearSelection();
}
```

Purpose: Adds product to cart.

3.1 Boundary Value Testing

Test Case ID	Input State	Expected Output	Test Type	Reason
BVT-ADD-1	No product selected (selectedRowIndex = -1)	Error message appears	Boundary	Invalid selection
BVT-ADD-2	Product selected, quantity = "1"	Item added to cart	Boundary	Minimum quantity
BVT-ADD-3	Product selected, quantity = "0"	Error/not added	Boundary	Invalid quantity
BVT-ADD-4	Product selected, quantity = "999"	Item added	Robustness	Large quantity
BVT-ADD-5	Product selected, quantity = "1.5"	Error/not added	Invalid	Non-integer quantity
BVT-ADD-6	Product selected, quantity = "-5"	Error/not added	Invalid	Negative quantity
BVT-ADD-7	Product selected, quantity = "abc"	Error message	Invalid	Non-numeric input

- This BVT tests selection states (selected/not selected) and quantity boundaries (valid/integer/positive). The method handles basic cases, but lacks validation for decimal/negative quantities which may cause calculation errors. Input validation before parseInt() would improve robustness.

3.2 Decision Table Testing

Condition\Rules	R1	R2	R3	R4	R5	R6
C1: Product selected?	No	Yes	Yes	Yes	Yes	Yes
C2: Quantity numeric?	N/A	No	Yes	Yes	Yes	Yes
C3: Quantity > 0?	N/A	N/A	No	Yes	Yes	Yes
C4: Quantity integer?	N/A	N/A	N/A	No	Yes	Yes
C5: Valid price?	N/A	N/A	N/A	N/A	Yes	No
Action: Show error message	✓	✓	✓			✓
Action: Add to cart table				✓	✓	
Action: Calculate total				✓	✓	
Action: Clear selection	✓	✓	✓	✓	✓	✓

The decision table reveals that the method only works correctly under ideal conditions (Rule 5) and fails or crashes in most invalid input scenarios, particularly for non-numeric, non-integer, or non-positive quantities highlighting missing validation and error recovery logic that leaves the system vulnerable to user input errors.

3.3 Code Coverage Analysis

Coverage Type	Percentage	Covered Elements	Uncovered Elements
Statement Coverage	~85%	• Product selection check • Table addition • Total calculation • Selection clearing	• Error handling for non-integer quantities (NumberFormatException not caught)
Branch Coverage	~75%	• if (selectedRowIndex != -1) → True/False paths • Error message display branch	• Exception handling for Integer.parseInt() failures • Input validation before parsing
Path Coverage	~60%	Covered Paths: 1. No selection → error message 2. Valid selection + valid integer → success	Uncovered Paths: 1. Valid selection + non-numeric quantity → exception thrown 2. Valid selection + decimal quantity → exception thrown 3. Valid selection + empty quantity → exception thrown

5. TEST RESULTS SUMMARY

1. Overall Test Effectiveness: MODERATE

Both methods exhibit partial functionality but suffer from incomplete error handling and insufficient input validation, resulting in a system that works under ideal conditions, but fails unpredictably with real world user input.

2. Functional Completeness: PARTIAL

- **calculateTotal(): Works correctly for valid inputs but lacks error recovery mechanisms**

- **addToCartBtnActionPerformed(): Basic functionality present but crashes on invalid input**
- **Critical gap: No graceful degradation when users enter non-numeric or malformed data**

3. Robustness Assessment: POOR

- **7 out of 14 BVT cases likely to fail due to unhandled exceptions**
- **Both methods use direct parsing without validation (Double.parseDouble(), Integer.parseInt())**
- **No input sanitization (whitespace, empty strings, negative values)**
- **Decision tables reveal missing error states in business logic**

4. Test Coverage Status: INCOMPLETE

- **Statement coverage: ~80-85% (basic paths covered)**
- **Branch coverage: ~70-75% (error paths missing)**
- **Path coverage: ~60% (exception flows untested)**
- **MC/DC achievable but not implemented due to missing validation logic**

5. Risk Assessment: HIGH

- **High probability of runtime crashes in production**
- **Data corruption risk from unvalidated calculations**
- **Poor user experience with cryptic error messages**
- **Cascading failures likely (broken cart → broken sales → broken invoices)**

CONCLUSION: The core calculation logic is sound, but the absence of defensive programming makes the system fragile. Immediate fixes should focus on adding input validation, exception handling, and comprehensive error recovery before deployment.