**Sarah Lorenzen**

**Process Summary:** Simple Calculator Implementation

**1. Project Overview**

This project involved designing and implementing a command-line calculator application in Python with comprehensive unit testing. The calculator performs four basic arithmetic operations (addition, subtraction, multiplication, and division) with robust error handling and user-friendly interaction patterns.

**2. Development Approach & Strategy**

*Initial Analysis:*

Reviewed requirements to identify core functionality: menu display, input validation, arithmetic operations, error handling, and program flow control. Created flowchart to visualize program logic and identify decision points. Wrote pseudocode to plan control structures before implementation.

*Design Decisions:*

Modular architecture: Separated concerns into individual functions (display, input, operations)

Defensive programming: Implemented input validation at every user interaction point

Error handling first: Planned for edge cases (division by zero, invalid input) from the start

Test-driven mindset: Designed functions to be easily testable with clear inputs/outputs

*Implementation Phase 1: Core Functions*

* Implemented arithmetic functions (add, subtract, multiply, divide)
* Each function returns a clear result or None for errors
* Kept functions deterministic and without side effects for easier testing

*Implementation Phase 2: User Interface*

* Created display\_menu() for consistent UI
* Implemented get\_operation\_choice() with validation loop
* Built get\_number() helper with try-except for robust input handling

*Implementation Phase 3: Program Flow*

* Implemented main() loop with clear exit conditions
* Added perform\_calculation() to route operations
* Integrated error messages and result display

*Phase 4: Testing*

* Developed **comprehensive** unit test suite
* Organized tests by operation type
* Added edge case testing

**3. Issues Encountered & Resolutions**

*Issue #1: Division by Zero*

Problem: Initial implementation would raise ZeroDivisionError when user entered 0 as divisor, causing program crash.

Resolution:

def divide(num1, num2):

if num2 == 0:

return None # Graceful handling

return num1 / num2

Result: Program now returns None for division by zero, which is checked in main loop and displays a user-friendly error message instead of crashing.

Testing:

def test\_divide\_by\_zero(self):

self.assertIsNone(divide(10, 0))

self.assertIsNone(divide(0, 0))

All division by zero tests pass (4/4)

*Issue #2: Invalid User Input*

Problem: Users could enter non-numeric values for numbers or invalid menu choices, causing ValueError exceptions.

Resolution:

def get\_number(prompt):

while True:

try:

return float(input(prompt))

except ValueError:

print("Error: Invalid input. Please enter a valid number.")

Strategy:

* Used try-except blocks to catch ValueError
* Implemented validation loops that continue until valid input received
* Provided clear error messages guiding user to correct their input

Result: Program won’t crash from invalid input; users get helpful feedback and a chance to retry

*Issue #3: Code Reusability*

Problem: Design without repeated validation code in multiple locations.

Resolution: Extracted common validation logic into reusable helper functions

get\_operation\_choice() handles all menu validation

get\_number() handles all numeric input validation

Benefits:

* DRY (Don't Repeat Yourself) principle
* Easier to maintain and update validation rules

**4. Current Code Optimization for Performance**

Early Returns: Avoids unnecessary computation when error detected

def divide(num1, num2):

if num2 == 0:

return None # Exit immediately on error

return num1 / num2

Minimal Function Calls: Direct arithmetic operations (no unnecessary abstractions)

def add(num1, num2):

return num1 + num2

Efficient Input Validation: Single-pass validation with short-circuit evaluation.

if 1 <= choice <= 5:

return choice

String Formatting Optimization: f-strings (faster than .format() or % formatting)

**5. Potential Future Optimizations**

*Optimization 1: Expression Evaluation*

Current: Only handles two numbers per operation

Improvement: Add expression parser for complex calculations

*Optimization 2: Calculation History*

Current: No memory of previous calculations

Improvement: Store history in memory for recall

**6. Key Takeaways/Lessons Learned**

* Input validation is critical: Never trust user input
* Edge cases matter: Division by zero would crash without explicit handling
* Modular design benefits: Testing is much easier with small, focused functions
* Error messages should guide: Tell users what went wrong and how to fix it
* Design before code: Flowchart and pseudocode saved time
* Test early, test often: Writing tests alongside code catches bugs faster
* User experience matters: Clear prompts and formatting improve usability