Name: Asjad Jadoon

FA22-BSE-129

Problems in Software

1. Incorrect Calculations

This happens when the software produces wrong results in functions that involve numbers or dates. For example, a financial program might miscalculate interest rates, or a date function could incorrectly calculate the number of days between two dates.

- Software facing this problem
  - QuickBooks (accounting software)
  - Excel (spreadsheet software)
  - SAP (financial software)
  - DateCalc (date calculation software)

2. Incorrect Data Edits

This refers to when the software's rules for data entry are not applied correctly. For example, it might let you enter a date like "February 30," even though that's not a valid date, because the system only checks for day limits (1-31) and not the actual month.

- **Software facing this problem**
  - Google Forms (data entry forms)
  - Salesforce (CRM system)
  - Microsoft Access (database management system)
  - HR systems (e.g., ADP Workforce Now)

3. <u>Ineffective Data Edits</u>

Sometimes the software has data validation in place, but it doesn’t catch all possible issues. For example, if a user enters an address with spaces before the numbers or letters, the system might still accept it, causing problems later when searching or sorting addresses.

- **Software facing this problem**
  - Amazon (e-commerce website—issues with product searches if data entry isn't cleaned)
  - Microsoft Dynamics (CRM system)
  - Magento (e-commerce platform)
  - Mailchimp (email marketing software)

4. <u>Incorrect Business Rules</u>

This is when the software doesn't behave as expected because the business rules (the instructions on how the software should function) were either misunderstood or misapplied. This can happen when the system’s developers interpret the requirements wrongly or the requirements were vague or incomplete.

- **Software facing this problem**
  - Oracle ERP (enterprise resource planning system)
  - Workday (HR and finance management software)

- PeachTree (business accounting software)
- Shopify (e-commerce platform, when business rules for transactions or taxes are incorrectly configured)

5. <u>Poor Software Performance</u>

This issue happens when the software works slowly, takes too long to respond, or can't handle a lot of data or users at once. For example, a website might load very slowly, or a database might take too long to retrieve information due to performance issues.

- Software facing this problem
- Zoom (video conferencing platform—issues with lag during high traffic times)
- Salesforce (CRM system with large datasets)
- Amazon Web Services (AWS) (cloud services, which may face performance degradation under heavy loads)
- JIRA (project management tool when handling large projects or high user counts)

<u>Solutions of Problem</u>

<u>(Incorrect Calculations)</u>

1. Thorough Testing and Validation

Solution: Implement robust unit testing and validation techniques to catch calculation errors.

Automated Testing: Develop automated tests that simulate real-world inputs and test the calculations thoroughly. For instance, if you are dealing with financial calculations, test for edge cases like leap years, interest rate variations, etc.

Boundary Testing: Test the software with boundary values (e.g., maximum and minimum values) to ensure calculations handle extreme cases correctly.

Regression Testing: Regularly test existing functionality after updates or changes to ensure previously correct calculations remain intact.

2. Clear and Accurate Requirements

Solution: Ensure that the business rules, formulas, and calculations are correctly defined and understood.

Consult Subject Matter Experts (SMEs): In the case of financial software, work closely with financial experts to ensure that formulas, tax rates, interest calculations, etc., are implemented correctly.

Documentation of Business Rules: Clearly document the formulas, rules, and algorithms that need to be implemented. Misunderstanding the business logic or misinterpretation of requirements can often lead to calculation errors.

Review Requirements Regularly: Regularly review and validate the calculations against updated business rules or regulations.

3. Use of Established Libraries and Frameworks

Solution: Utilize well-tested libraries and frameworks to handle complex calculations rather than writing custom code.

Financial Libraries: In accounting or financial software, use established libraries (like the Decimal module in Python for precise floating-point calculations) to handle calculations more accurately.

Date Libraries: For date calculations, libraries like Moment.js (JavaScript) or DateTime (Python) can manage complex date manipulations like leap years, daylight saving time adjustments, and month lengths.

Mathematical Libraries: For mathematical operations, consider using libraries like NumPy (for Python) or Apache Commons Math (for Java) to ensure calculations are performed correctly with high precision.

4. Precision Handling

Solution: Ensure high precision in mathematical and financial calculations to avoid rounding errors or floating-point inaccuracies.

Avoid Floating-Point for Financial Calculations: Use arbitrary precision libraries like BigDecimal (Java) or Decimal (Python) to avoid the rounding issues associated with floating-point numbers.

Round Appropriately: Implement proper rounding techniques, particularly in financial systems, to ensure that results are rounded to the correct number of decimal places according to the specific business rules (e.g., rounding currency values to two decimal places).

5. User Input Validation

Solution: Validate inputs to ensure that incorrect or invalid data doesn't lead to incorrect calculations.

Field-Level Validation: Before performing calculations, validate user inputs (e.g., amounts, dates, rates) to ensure they fall within acceptable ranges. For example, check that interest rates are within a valid range or that dates are properly formatted and realistic (e.g., no negative values or impossible dates like February 30th).

Input Sanitization: Sanitize inputs to remove or prevent any invalid characters that might disrupt calculations or lead to errors (e.g., non-numeric input in fields that should only accept numbers).

6. Handling Edge Cases

Solution: Account for edge cases and exceptional scenarios in the calculation logic.

Financial Calculations: Ensure that edge cases such as leap years, different tax rates, and varying month lengths are handled appropriately. For example, a monthly interest calculation system should consider how many days are in each month when calculating prorated values.

Date Calculations: Take into account edge cases like time zone differences, daylight savings time, and leap years when working with date calculations.

Boundary Cases: Always test for boundary conditions, like the maximum possible values, zero, or negative values.

7. Error Handling and Logging

Solution: Implement proper error handling and logging mechanisms to track and debug calculation errors.

Error Detection: Design the system to detect invalid calculations, such as division by zero or overflow errors. Return appropriate error messages or default values (if necessary) rather than allowing incorrect calculations to propagate.

Logging: Implement logging to capture failed calculations, incorrect inputs, and issues in the calculation process. This helps identify the root cause of the issue and track down bugs more easily.

Audit Trails: In systems that require high accuracy (e.g., financial or medical software), maintain an audit trail of calculations, including input data, formulas used, and the final result.

8. Regular Review and Updates

Solution: Ensure ongoing monitoring and maintenance of the software, especially when business rules or formulas change.

Periodic Reviews: As business rules evolve or tax laws change, regularly update the software's calculation logic to stay in compliance.

Software Patches: If bugs or errors are found after deployment, ensure the team is able to deploy fixes quickly and test the patched calculations to ensure they are now correct.

9. User Training and Documentation

Solution: Provide proper training and clear documentation to users to ensure they understand how calculations are performed and avoid user error.

Training: Users should be educated on how to input data correctly and how calculations are performed. For example, in financial software, users need to understand how to input rates and periods accurately.

Documentation: Provide detailed user manuals or tooltips that explain the calculation formulas and guidelines. This reduces user error and helps ensure that the system is being used as intended.

10. Consultation with Domain Experts

Solution: Regularly consult with domain experts to ensure that business rules and formulas are implemented correctly.

Subject Matter Experts (SMEs): Work with professionals in the field (e.g., accountants for financial software, legal experts for contract software) to validate that all business logic aligns with industry standards.

Verification: Before finalizing any system, have these experts review the software’s calculation logic and ensure it reflects real-world requirements accurately.

Code

```
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
import java.util.logging.Logger;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculationHandler {
    // Logger for error tracking
    private static final Logger logger =
Logger.getLogger(CalculationHandler.class.getName());

    // Method to calculate interest using BigDecimal for precision
    public BigDecimal calculateInterest(BigDecimal principal, BigDecimal
rate, int years) {
        BigDecimal onePlusRate = BigDecimal.ONE.add(rate);
        BigDecimal result = principal.multiply(onePlusRate.pow(years));
        return result.setScale(2, RoundingMode.HALF_UP); // Round to two
decimal places
    }

    // Method to calculate the number of days between two dates
    public long calculateDaysBetweenDates(LocalDate start, LocalDate
end) {
        return ChronoUnit.DAYS.between(start, end);
    }

    // Method to calculate tax with validation
    public double calculateTax(double income, double taxRate) {
        if (income <= 0 || taxRate < 0 || taxRate > 1) {
            throw new IllegalArgumentException("Invalid income or tax rate.");
        }
        return income * taxRate;
    }

    // Input validation for interest rate
    public void validateInterestRate(double rate) {
        if (rate < 0 || rate > 1) {
            throw new IllegalArgumentException("Interest rate must be between 0 and 1.");
        }
    }

    // Method to sanitize and clean user input (remove non-numeric characters)
    public String sanitizeInput(String input) {
        return input.replaceAll("[^0-9.]", ""); // Allow only numeric and decimal points
    }

    // Error handling and logging
    public double divide(double numerator, double denominator) {
        try {
            if (denominator == 0) {
                throw new ArithmeticException("Cannot divide by zero");
            }
            return numerator / denominator;
        } catch (ArithmeticException e) {
            logger.warning("Error: " + e.getMessage());
            return 0;  // Return a default value when an error occurs
        }
    }

    // Test for calculating interest using JUnit
    @Test
    public void testCalculateInterest() {
        BigDecimal principal = new BigDecimal("1000");
        BigDecimal rate = new BigDecimal("0.05");
        assertEquals(new BigDecimal("1276.28"), calculateInterest(principal,
rate, 5));
    }

    // Boundary testing to check edge cases
    @Test
    public void testBoundaryInterest() {
        assertEquals(new BigDecimal("1000.00"), calculateInterest(new BigDecimal("1000"), new BigDecimal("0"), 5));
    }

    // Test for calculating tax
    @Test
    public void testCalculateTax() {
        assertEquals(500.00, calculateTax(10000, 0.05), 0.01);
    }

    // Test for date calculations (Leap year case)
    @Test
    public void testCalculateDaysBetweenDates() {
        LocalDate start = LocalDate.of(2020, 2, 29); // Leap year
        LocalDate end = LocalDate.of(2021, 2, 28);  // One year later (not a leap year)
        assertEquals(366, calculateDaysBetweenDates(start, end));
    }

    // Test for division with error handling
    @Test
    public void testDivide() {
        assertEquals(2.0, divide(10, 5), 0.01);
        assertEquals(0.0, divide(10, 0), 0.01); // Division by zero
    }

    // Main method to demonstrate functionality
    public static void main(String[] args) {
        CalculationHandler handler = new CalculationHandler();

        // Calculate interest for financial calculation example
        BigDecimal principal = new BigDecimal("1000");
        BigDecimal rate = new BigDecimal("0.05");
        System.out.println("Interest: " + handler.calculateInterest(principal,
rate, 5));

        // Sanitize input example
        String sanitizedInput = handler.sanitizeInput("$1,000.50");
        System.out.println("Sanitized Input: " + sanitizedInput);

        // Tax calculation example
        System.out.println("Tax: " + handler.calculateTax(10000, 0.05));

        // Date calculation example (Leap year)
        LocalDate start = LocalDate.of(2020, 2, 29);

        LocalDate end = LocalDate.of(2021, 2, 28);

        System.out.println("Days between: " +
handler.calculateDaysBetweenDates(start, end));


        // Division example with error handling

        System.out.println("Division: " + handler.divide(10, 2));

        System.out.println("Division by zero: " + handler.divide(10, 0));

    }

}
```