Team Member 3: CRUD Operations & Data Validation Specialist

Your Role

You ensure all data operations work correctly, add validation, handle errors gracefully, and test the entire system thoroughly.

Day 1 Tasks (5-7 hours)

Morning Session (3-4 hours)

1. Understand CRUD Operations

Concepts to learn: - **C**reate - Adding new records - **R**ead - Viewing/retrieving records - **U**pdate - Modifying existing records - **D**elete - Removing records

Why validation matters: - Prevents bad data from entering database - Improves user experience - Reduces bugs and crashes

Quick Tutorial:

```
# Example validation functions
def validate_username(username):
    """Check if username is valid"""
    if len(username) < 4:
        return False, "Username must be at least 4 characters"
    if not username.isalnum():
        return False, "Username can only contain letters and numbers"
    return True, "Valid"

def validate_email(email):
    """Check if email is valid"""
    if '@' not in email or '.' not in email:
        return False, "Invalid email format"
    return True, "Valid"

# Testing
```

```
is_valid, message = validate_username("ab")
print(message)  # "Username must be at least 4 characters"
```

2. Study Error Handling

Learn about try-except blocks:

```
try:
    # Code that might fail
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
except Exception as e:
    print(f"An error occurred: {e}")
finally:
    print("This always runs")
```

Afternoon Session (2-3 hours)

3. Implement Your Code

File: validation_utils.py

```
"""
Validation and Data Utilities Module
Handles input validation and data processing
Author: [Your Name] - Team Member 3
"""

import re
from datetime import datetime
from typing import Tuple, List, Any


class ValidationUtils:
    """Utility class for data validation"""

    # ============= USER INPUT VALIDATION =============

    @staticmethod
    def validate_username(username: str) -> Tuple[bool, str]:
        """
        Validate username format
        Returns: (is_valid: bool, message: str)
        """
        if not username:
            return False, "Username cannot be empty"
```

```
        if len(username) < 4:
            return False, "Username must be at least 4 characters long"

        if len(username) > 20:
            return False, "Username cannot exceed 20 characters"

        if not username[0].isalpha():
            return False, "Username must start with a letter"

        if not re.match(r'^[a-zA-Z0-9_]+$', username):
            return False, "Username can only contain letters, numbers, and underscores"

        return True, "Valid username"

    @staticmethod
    def validate_password(password: str) -> Tuple[bool, str]:
        """
        Validate password strength
        Requirements: at least 6 characters, contains letter and number
        Returns: (is_valid: bool, message: str)
        """
        if not password:
            return False, "Password cannot be empty"

        if len(password) < 6:
            return False, "Password must be at least 6 characters long"

        if len(password) > 50:
            return False, "Password cannot exceed 50 characters"

        has_letter = any(c.isalpha() for c in password)
        has_number = any(c.isdigit() for c in password)

        if not (has_letter and has_number):
            return False, "Password must contain both letters and numbers"

        return True, "Strong password"

    @staticmethod
    def validate_fullname(fullname: str) -> Tuple[bool, str]:
        """
        Validate full name format
        Returns: (is_valid: bool, message: str)
        """
        if not fullname:
            return False, "Full name cannot be empty"

        if len(fullname) < 2:
            return False, "Full name is too short"

        if len(fullname) > 50:
            return False, "Full name cannot exceed 50 characters"
```

```
        if not re.match(r'^[a-zA-Z\s\-\']+$', fullname):
            return False, "Full name can only contain letters, spaces, hyphens, and apo

        return True, "Valid full name"

    # ============= RECORD VALIDATION =============

    @staticmethod
    def validate_title(title: str) -> Tuple[bool, str]:
        """
        Validate record title
        Returns: (is_valid: bool, message: str)
        """
        if not title:
            return False, "Title cannot be empty"

        if len(title) < 3:
            return False, "Title must be at least 3 characters long"

        if len(title) > 100:
            return False, "Title cannot exceed 100 characters"

        return True, "Valid title"

    @staticmethod
    def validate_description(description: str) -> Tuple[bool, str]:
        """
        Validate record description
        Returns: (is_valid: bool, message: str)
        """
        if not description:
            return False, "Description cannot be empty"

        if len(description) < 10:
            return False, "Description must be at least 10 characters long"

        if len(description) > 500:
            return False, "Description cannot exceed 500 characters"

        return True, "Valid description"

    @staticmethod
    def validate_category(category: str, valid_categories: List[str] = None) -> Tuple[b
        """
        Validate category selection
        Returns: (is_valid: bool, message: str)
        """
        if not category:
            return False, "Category must be selected"

        if valid_categories and category not in valid_categories:
```

```
            return False, f"Category must be one of: {', '.join(valid_categories)}"

        return True, "Valid category"

    @staticmethod
    def validate_status(status: str) -> Tuple[bool, str]:
        """
        Validate record status
        Returns: (is_valid: bool, message: str)
        """
        valid_statuses = ["Active", "Inactive", "Completed"]

        if not status:
            return False, "Status must be selected"

        if status not in valid_statuses:
            return False, f"Status must be one of: {', '.join(valid_statuses)}"

        return True, "Valid status"

    # ============= COMPLETE VALIDATION =============

    @staticmethod
    def validate_user_registration(username: str, password: str, confirm_password: str,
        """
        Validate complete user registration data
        Returns: (is_valid: bool, message: str)
        """
        # Check username
        is_valid, message = ValidationUtils.validate_username(username)
        if not is_valid:
            return False, f"Username Error: {message}"

        # Check full name
        is_valid, message = ValidationUtils.validate_fullname(fullname)
        if not is_valid:
            return False, f"Full Name Error: {message}"

        # Check password
        is_valid, message = ValidationUtils.validate_password(password)
        if not is_valid:
            return False, f"Password Error: {message}"

        # Check password match
        if password != confirm_password:
            return False, "Passwords do not match"

        return True, "All fields valid"

    @staticmethod
    def validate_record_data(title: str, description: str, category: str) -> Tuple[bool
        """
```

```
        Validate complete record data
        Returns: (is_valid: bool, message: str)
        """
        # Check title
        is_valid, message = ValidationUtils.validate_title(title)
        if not is_valid:
            return False, f"Title Error: {message}"

        # Check description
        is_valid, message = ValidationUtils.validate_description(description)
        if not is_valid:
            return False, f"Description Error: {message}"

        # Check category
        valid_categories = ["General", "Important", "Personal", "Work", "Other"]
        is_valid, message = ValidationUtils.validate_category(category, valid_categories)
        if not is_valid:
            return False, f"Category Error: {message}"

        return True, "All fields valid"


class DataFormatter:
    """Utility class for formatting data for display"""

    @staticmethod
    def format_date(date_string: str, format_type: str = "short") -> str:
        """
        Format date string for display
        format_type: 'short' (YYYY-MM-DD) or 'long' (January 15, 2024)
        """
        try:
            if isinstance(date_string, str):
                # Parse the date string
                dt = datetime.fromisoformat(date_string.replace('Z', '+00:00'))
            else:
                dt = date_string

            if format_type == "short":
                return dt.strftime("%Y-%m-%d")
            elif format_type == "long":
                return dt.strftime("%B %d, %Y")
            elif format_type == "time":
                return dt.strftime("%Y-%m-%d %H:%M:%S")
            else:
                return str(date_string)

        except Exception as e:
            return str(date_string)

    @staticmethod
    def truncate_text(text: str, max_length: int = 50) -> str:
```

```
        """Truncate long text with ellipsis"""
        if len(text) <= max_length:
            return text
        return text[:max_length-3] + "..."

    @staticmethod
    def format_status(status: str) -> str:
        """Format status with emoji"""
        status_map = {
            "Active": "✓ Active",
            "Inactive": "○ Inactive",
            "Completed": "✔ Completed"
        }
        return status_map.get(status, status)

    @staticmethod
    def sanitize_input(text: str) -> str:
        """Remove potentially dangerous characters from input"""
        # Remove any HTML-like tags
        text = re.sub(r'<[^>]+>', '', text)
        # Remove excessive whitespace
        text = ' '.join(text.split())
        return text.strip()


class ErrorHandler:
    """Utility class for consistent error handling"""

    @staticmethod
    def handle_database_error(error: Exception) -> str:
        """Convert database errors to user-friendly messages"""
        error_str = str(error).lower()

        if "unique constraint" in error_str:
            return "This record already exists"
        elif "not null constraint" in error_str:
            return "Required field is missing"
        elif "foreign key constraint" in error_str:
            return "Cannot delete: record is referenced elsewhere"
        elif "no such table" in error_str:
            return "Database not properly initialized"
        elif "database is locked" in error_str:
            return "Database is busy, please try again"
        else:
            return f"Database error: {str(error)}"

    @staticmethod
    def handle_validation_error(field_name: str, error_message: str) -> str:
        """Format validation error messages consistently"""
        return f"{field_name}: {error_message}"

    @staticmethod
```

```
    def log_error(error: Exception, context: str = ""):
        """Log errors for debugging"""
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        log_message = f"[{timestamp}] {context}: {type(error).__name__} - {str(error)}\

        try:
            with open("error_log.txt", "a") as f:
                f.write(log_message)
        except:
            print(log_message)


# ============= TESTING CODE =============
if __name__ == "__main__":
    print("=== Validation Testing ===\n")

    # Test 1: Username validation
    print("Test 1: Username Validation")
    test_usernames = ["abc", "user123", "test_user", "ab", "user@123", "a"*25]
    for username in test_usernames:
        is_valid, message = ValidationUtils.validate_username(username)
        print(f"  '{username}': {message}")

    print("\nTest 2: Password Validation")
    test_passwords = ["12345", "abc123", "password", "pass", "StrongPass123"]
    for password in test_passwords:
        is_valid, message = ValidationUtils.validate_password(password)
        print(f"  '{password}': {message}")

    print("\nTest 3: Record Validation")
    test_records = [
        ("", "Description", "General"),
        ("AB", "Short description", "General"),
        ("Valid Title", "This is a valid description with enough characters", "General"
        ("Valid Title", "Short", "General"),
    ]
    for title, desc, cat in test_records:
        is_valid, message = ValidationUtils.validate_record_data(title, desc, cat)
        print(f"  Title: '{title[:20]}...': {message}")

    print("\nTest 4: Date Formatting")
    test_date = "2024-01-15 10:30:45"
    print(f"  Short: {DataFormatter.format_date(test_date, 'short')}")
    print(f"  Long: {DataFormatter.format_date(test_date, 'long')}")
    print(f"  Time: {DataFormatter.format_date(test_date, 'time')}")

    print("\nTest 5: Text Truncation")
    long_text = "This is a very long text that should be truncated to show only the fir
    print(f"  Original ({len(long_text)} chars): {long_text}")
    print(f"  Truncated: {DataFormatter.truncate_text(long_text, 30)}")

    print("\nTest 6: Input Sanitization")
```

```
    dirty_input = "  Text with <script>alert('xss')</script>  extra   spaces  "
    clean = DataFormatter.sanitize_input(dirty_input)
    print(f"  Original: '{dirty_input}'")
    print(f"  Cleaned: '{clean}'")

    print("\n✓ All tests completed!")
```

File: integration_tests.py

```
"""
Integration Testing Module
Tests the complete system workflow
Author: [Your Name] - Team Member 3
"""

import sys
from typing import List, Tuple

def test_database_integration():
    """Test database operations"""
    print("\n=== Database Integration Test ===")

    try:
        from database_manager import DatabaseManager
        db = DatabaseManager()

        # Test 1: Create user
        print("\n1. Testing user creation...")
        success, msg = db.create_user("testuser123", "testpass123", "Test User")
        print(f"   Result: {msg}")
        assert success or "already exists" in msg.lower()

        # Test 2: Authenticate
        print("\n2. Testing authentication...")
        success, user_id, msg = db.authenticate_user("testuser123", "testpass123")
        print(f"   Result: {msg}")
        assert success

        # Test 3: Create record
        print("\n3. Testing record creation...")
        success, msg = db.create_record(
            user_id,
            "Test Record",
            "This is a test record for integration testing",
            "General"
        )
        print(f"   Result: {msg}")
        assert success

        # Test 4: Read records
```

```
        print("\n4. Testing record reading...")
        records = db.read_all_records(user_id)
        print(f"   Found {len(records)} record(s)")
        assert len(records) > 0

        # Test 5: Update record
        if records:
            print("\n5. Testing record update...")
            record_id = records[0][0]
            success, msg = db.update_record(
                record_id,
                "Updated Title",
                "Updated description",
                "Important",
                "Active"
            )
            print(f"   Result: {msg}")
            assert success

        # Test 6: Search records
        print("\n6. Testing search functionality...")
        results = db.search_records(user_id, "Updated")
        print(f"   Found {len(results)} matching record(s)")
        assert len(results) > 0

        # Test 7: Get statistics
        print("\n7. Testing statistics...")
        stats = db.get_summary_stats(user_id)
        print(f"   Total records: {stats['total']}")
        print(f"   Active: {stats['active']}")
        print(f"   Categories: {stats['by_category']}")

        print("\n✓ All database tests passed!")
        return True

    except ImportError:
        print("✗ Database module not found")
        return False
    except Exception as e:
        print(f"✗ Test failed: {e}")
        return False

def test_validation_integration():
    """Test validation with actual data"""
    print("\n=== Validation Integration Test ===")

    try:
        from validation_utils import ValidationUtils

        # Test registration validation
        print("\n1. Testing complete registration validation...")
        is_valid, msg = ValidationUtils.validate_user_registration(
```

```
            "newuser",
            "pass123",
            "pass123",
            "New User"
        )
        print(f"   Result: {msg}")
        assert is_valid

        # Test with mismatched passwords
        print("\n2. Testing password mismatch detection...")
        is_valid, msg = ValidationUtils.validate_user_registration(
            "newuser",
            "pass123",
            "pass456",
            "New User"
        )
        print(f"   Result: {msg}")
        assert not is_valid
        assert "match" in msg.lower()

        # Test record validation
        print("\n3. Testing record validation...")
        is_valid, msg = ValidationUtils.validate_record_data(
            "Valid Record Title",
            "This is a valid description with sufficient length",
            "General"
        )
        print(f"   Result: {msg}")
        assert is_valid

        print("\n✓ All validation tests passed!")
        return True

    except ImportError:
        print("✗ Validation module not found")
        return False
    except Exception as e:
        print(f"✗ Test failed: {e}")
        return False

def test_full_workflow():
    """Test complete user workflow"""
    print("\n=== Full Workflow Test ===")

    try:
        from database_manager import DatabaseManager
        from validation_utils import ValidationUtils, DataFormatter

        db = DatabaseManager()

        # Step 1: Validate and create user
        print("\n1. Complete user registration workflow...")
```

```
        username = "workflow_test"
        password = "test123"
        fullname = "Workflow Tester"

        is_valid, msg = ValidationUtils.validate_user_registration(
            username, password, password, fullname
        )

        if is_valid:
            success, msg = db.create_user(username, password, fullname)
            print(f"   Registration: {msg}")

        # Step 2: Login
        print("\n2. Login workflow...")
        success, user_id, msg = db.authenticate_user(username, password)
        print(f"   Login: {msg}")
        assert success

        # Step 3: Validate and create record
        print("\n3. Complete record creation workflow...")
        title = "Workflow Test Record"
        description = "This record tests the complete workflow from validation to datab
        category = "General"

        is_valid, msg = ValidationUtils.validate_record_data(title, description, catego

        if is_valid:
            success, msg = db.create_record(user_id, title, description, category)
            print(f"   Record creation: {msg}")

        # Step 4: Retrieve and format
        print("\n4. Data retrieval and formatting...")
        records = db.read_all_records(user_id)
        if records:
            record = records[0]
            formatted_date = DataFormatter.format_date(record[4])
            truncated_desc = DataFormatter.truncate_text(record[2], 30)
            print(f"   Title: {record[1]}")
            print(f"   Description: {truncated_desc}")
            print(f"   Date: {formatted_date}")

        print("\n✓ Full workflow test passed!")
        return True

    except Exception as e:
        print(f"✗ Workflow test failed: {e}")
        import traceback
        traceback.print_exc()
        return False

def run_all_tests():
    """Run all integration tests"""
```

```
    print("=" * 50)
    print("SMART RECORDS SYSTEM - INTEGRATION TESTS")
    print("=" * 50)

    results = []

    # Run each test
    results.append(("Database Integration", test_database_integration()))
    results.append(("Validation Integration", test_validation_integration()))
    results.append(("Full Workflow", test_full_workflow()))

    # Summary
    print("\n" + "=" * 50)
    print("TEST SUMMARY")
    print("=" * 50)

    passed = sum(1 for _, result in results if result)
    total = len(results)

    for test_name, result in results:
        status = "✓ PASSED" if result else "✗ FAILED"
        print(f"{test_name:.<40}{status}")

    print(f"\nTotal: {passed}/{total} tests passed")

    if passed == total:
        print("\n🎉 All tests passed! System is ready.")
    else:
        print("\n⚠️  Some tests failed. Please review errors above.")

    return passed == total

if __name__ == "__main__":
    run_all_tests()
```

Day 2 Tasks (4-6 hours)

Morning Session (2-3 hours)

4. Run All Tests

Execute your test files:

```
# Test validation
python validation_utils.py
```

```
# Test integration
python integration_tests.py
```

Document any failures and fix them.

5. Add Enhanced Validation to GUI

Work with Team Member 2 to add validation to forms.

Example integration in their GUI code:

```
from validation_utils import ValidationUtils

def save_record(self):
    title = self.title_entry.get()
    description = self.description_text.get('1.0', tk.END)
    category = self.category_var.get()

    # ADD THIS VALIDATION
    is_valid, message = ValidationUtils.validate_record_data(
        title, description, category
    )

    if not is_valid:
        messagebox.showerror("Validation Error", message)
        return

    # Proceed with saving...
```

Afternoon Session (2-3 hours)

6. Create Test Documentation

File: TESTING_REPORT.md

```
# Testing Report - Smart Records System

## Test Summary
Date: [Current Date]
Tester: [Your Name]
Total Tests: [Number]
Passed: [Number]
Failed: [Number]

## Database Tests
✓ User creation
✓ User authentication
✓ Record creation
```

```
✓ Record reading
✓ Record updating
✓ Record deletion
✓ Search functionality
✓ Statistics generation

## Validation Tests
✓ Username validation
✓ Password validation
✓ Record data validation
✓ Input sanitization

## Integration Tests
✓ Complete registration workflow
✓ Complete login workflow
✓ Complete CRUD workflow

## Known Issues
1. [List any issues found]
2. [And how they were fixed]

## Browser/Environment Testing
- Operating System: [Windows/Mac/Linux]
- Python Version: [Version]
- All features working: Yes/No
```

7. Help Team with Integration

- Assist Team Member 2 with adding your validation to their GUI
- Verify Team Member 1's database handles errors properly
- Help Team Member 4 if they need data for reports

Checklist

Day 1

- ☐ Understand CRUD concepts
- ☐ Learn validation techniques
- ☐ Study error handling
- ☐ Write validation_utils.py
- ☐ Write integration_tests.py

- ☐ Test individual functions

Day 2

- ☐ Run full integration tests
- ☐ Document all test results
- ☐ Add validation to GUI forms
- ☐ Test complete workflows
- ☐ Create testing report
- ☐ Help team with integration

Things to Search and Learn

1. **Input Validation**: "Python input validation best practices"
2. **Regular Expressions**: "Python regex for text validation"
3. **Error Handling**: "Python try except finally explained"
4. **Testing**: "Python unit testing tutorial"
5. **Data Sanitization**: "How to prevent XSS attacks in Python"

Common Issues & Solutions

Issue: Validation too strict, rejecting valid input **Solution**: Review regex patterns, adjust length limits

Issue: Error messages not clear **Solution**: Make messages specific and actionable

Issue: Tests failing randomly **Solution**: Ensure test data is cleaned up between runs

Your Contribution (for presentation)

"I implemented all data validation to ensure only clean, correct data enters our system. I also created comprehensive tests to verify everything works together properly. The validation prevents errors and gives users helpful feedback."

Mark Breakdown for Your Work

- Database Integration & Accuracy: Shared with Member 1
- Code Quality & Documentation: 2 marks
- Total: 2 marks (13% of project)

Integration Points with Team

With Member 1 (Database): - Use their DatabaseManager class - Test all CRUD operations - Verify error handling

With Member 2 (GUI): - Add validation to their forms - Provide error messages - Test user workflows

With Member 4 (Reports): - Ensure data formatting is correct - Verify statistics calculations - Test report generation

Sample Integration Code

For GUI (give this to Member 2):

```
# In their signup window
from validation_utils import ValidationUtils

def create_account(self):
    # Get all inputs
    fullname = self.fullname_entry.get().strip()
    username = self.username_entry.get().strip()
    password = self.password_entry.get()
    confirm = self.confirm_entry.get()

    # VALIDATE EVERYTHING
    is_valid, message = ValidationUtils.validate_user_registration(
        username, password, confirm, fullname
    )

    if not is_valid:
        messagebox.showerror("Validation Error", message)
        return

    # Continue with database creation...
```

This ensures your validation is actually used!