Software Testing Assignment 1

Software Verification & Validation

Faizan Ahmad
(22F-3856)

Barira Aurangzeb
(22F-3316)

Section 8E

February 15, 2026

1 Phase A: Structural Analysis (White-Box Testing)

1.1 Control Flow Graph (CFG) - Pagination Logic

A Control Flow Graph (CFG) was constructed for the pagination logic implemented in the system. The CFG represents the internal control flow of the method responsible for splitting content into pages based on word limits.

The CFG includes:

- Initialization of variables
- Decision point for null or empty input
- Loop condition for iterating over content
- Decision for page break when page size limit is reached
- Normal and early return paths

All nodes and edges are labeled to clearly represent the control flow and decision points in the algorithm.

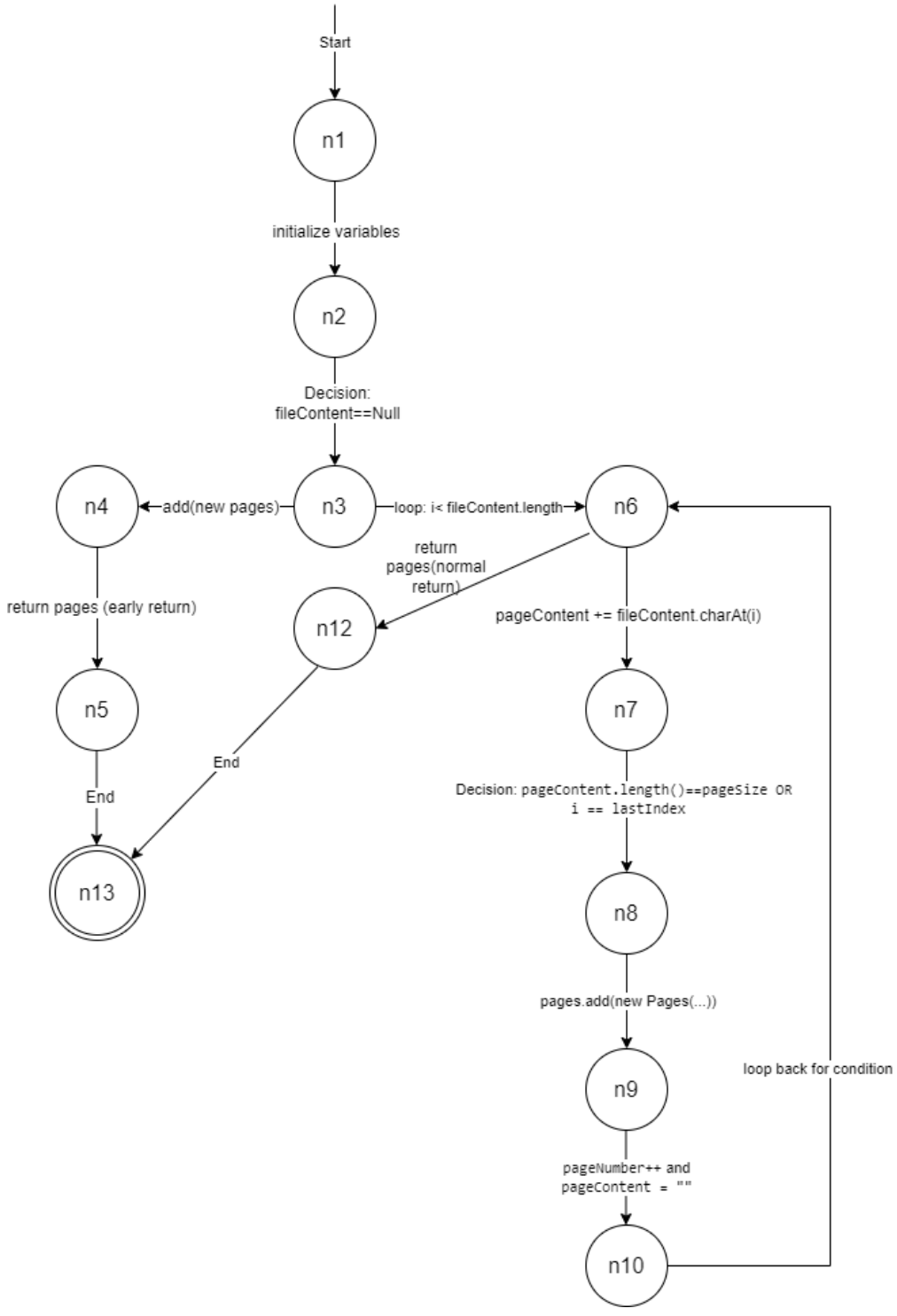

Figure 1: Control Flow Graph (CFG) for Pagination Logic

1.2 Cyclomatic Complexity - Pagination Logic

Cyclomatic complexity is calculated using the formula:

$$V(G) = E - N + 2P$$

Where:

- E = Number of edges
- N = Number of nodes
- P = Number of connected components

For the pagination logic CFG:

- $N = 12$
- $E = 13$
- $P = 1$

$$V(G) = 13 - 12 + 2(1) = 3$$

The cyclomatic complexity of the pagination logic is **3**. This means there are three independent execution paths that must be tested to achieve complete branch coverage.

1.3 Independent Test Paths - Pagination Logic

The set of independent paths is defined as:

$$P = \{p_1, p_2, p_3\}$$

Each path is represented as a tuple of CFG nodes:

$$p_1 = \langle n_1, n_2, n_3, n_4, n_5, n_{13} \rangle$$

(Early return when input is null or empty)

$$p_2 = \langle n_1, n_2, n_3, n_6, n_7, n_8, n_9, n_{10}, n_6, n_{12}, n_{13} \rangle$$

(Normal pagination with page break and loop back)

$$p_3 = \langle n_1, n_2, n_3, n_6, n_{12}, n_{13} \rangle$$

(Normal execution without triggering page break condition)

1.4 Auto-Save Trigger Analysis

1.4.1 Control Flow Graph (CFG)

A Control Flow Graph was constructed for the Auto-Save functionality, which automatically saves files when content exceeds 500 words.

The CFG includes:

- Selected row validation
- Field retrieval from table model
- File name extraction
- Content validation (null/empty check)
- Business logic call for file update
- Success/failure branching
- Status label updates
- Thread sleep mechanism
- Error logging and display

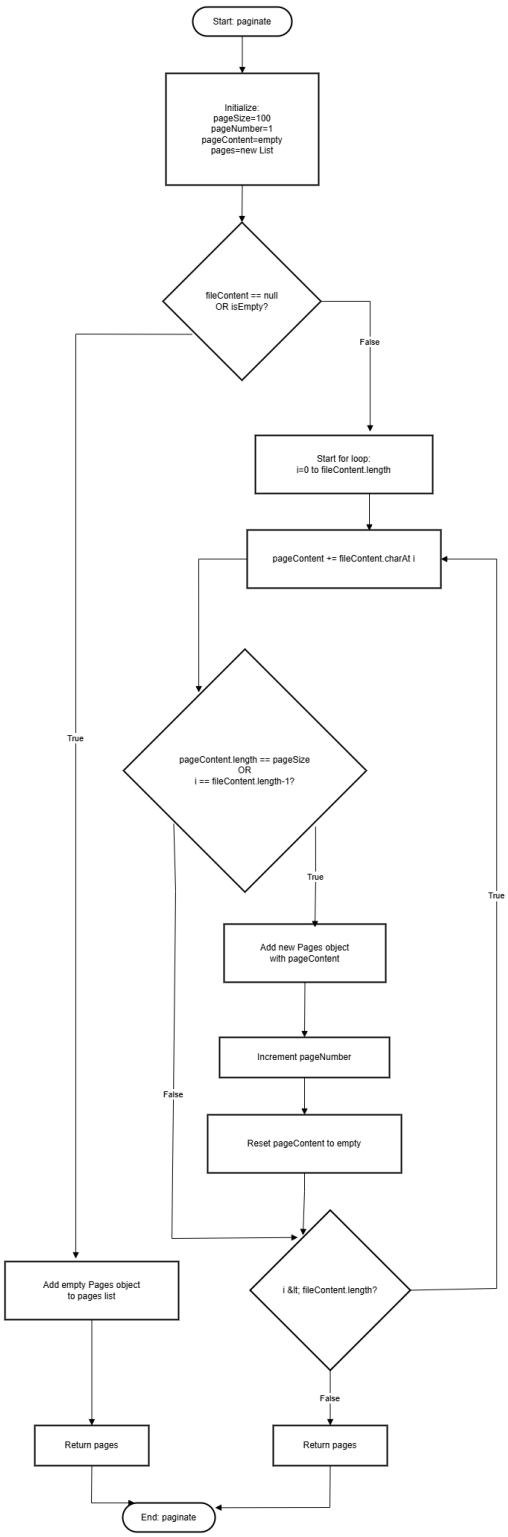

Figure 2: Control Flow Graph for Auto-Save Trigger Logic

1.4.2 Cyclomatic Complexity

Cyclomatic complexity is calculated using the formula:

$$V(G) = E - N + 2P$$

Where:

- E = Number of edges = 17
- N = Number of nodes = 15
- P = Number of connected components = 1

$$V(G) = 17 - 15 + 2(1) = 4$$

The cyclomatic complexity of the auto-save trigger logic is **4**. This indicates four independent execution paths that must be tested for complete branch coverage.

1.4.3 Independent Test Paths (Set Theory)

The set of independent paths is defined as:

$$P = \{p_1, p_2, p_3, p_4\}$$

Each path is represented as a tuple of CFG nodes:

$$p_1 = \langle n_1, n_2, n_{13}, n_{14}, n_{15} \rangle$$

(Early return when no row is selected)

$$p_2 = \langle n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9, n_{14}, n_{15} \rangle$$

(Content is null or empty, error path)

$$p_3 = \langle n_1, n_2, n_3, n_4, n_5, n_6, n_8, n_9, n_{14}, n_{15} \rangle$$

(Update succeeds, status label updated)

$$p_4 = \langle n_1, n_2, n_3, n_4, n_5, n_6, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{15} \rangle$$

(Update fails, error displayed and logged)

2 Phase B: Modular JUnit Testing

This phase focuses on black-box and modular testing using JUnit. The objective is to validate the correctness of business logic and data persistence logic through Positive, Negative, and Boundary test cases. All test classes are implemented inside the `Testing` folder as required.

2.1 Business Layer Testing

2.1.1 Command Pattern

The assignment requires testing the following Command Pattern methods:

- ImportCommand.execute()
- ExportCommand.execute()
- TransliterateCommand.execute()

However, in the provided base code, these command classes are not implemented. Therefore, JUnit test cases could not be executed for the Command Pattern.

Justification:

> The Command Pattern test cases could not be implemented because the required command classes (`ImportCommand`, `ExportCommand`, and `TransliterateCommand`) were not present in the provided project code. Hence, there was no executable logic available for testing.

2.1.2 TF-IDF Algorithm Testing

The TF-IDF algorithm was tested using JUnit test cases implemented in the `TFIDFCalculatorTest.java` file. The tests were designed to cover Positive, Negative, and Boundary scenarios as required by the assignment.

Test Environment

- Testing Framework: JUnit 5
- Package: `Testing.business`
- Class Tested: `TFIDFCalculator`

2.1.3 Test Case Results

Passed Test Cases

- **Single Word Document (Boundary Test)**
  The input contains a single word. The TF-IDF algorithm successfully returns a finite value without throwing any exception. This confirms correct handling of boundary input.

- **Word Not Present in Corpus (Boundary Test)**
  When a word not present in the corpus is tested, the algorithm assigns a default IDF value and returns a valid TF-IDF score. This confirms graceful handling of unseen terms.

- **Empty Document (Negative Test)**
  An empty document does not cause the system to crash. The function returns a finite value, demonstrating graceful error handling.

Failed Test Cases

- **Positive Path: Known Document Manual Check**
  The assignment requires the TF-IDF score to match the manually calculated value within a tolerance of ± 0.01. However, the implemented TF-IDF algorithm applies preprocessing and smoothing techniques which differ from the classical TF-IDF formula. As a result, the computed score does not match the manually calculated value, causing this test to fail.

- **Special Characters Only (Negative Test)**
  When the input document contains only special characters, preprocessing removes all tokens, resulting in an empty word list. This leads to invalid TF-IDF computation and causes the test to fail. This highlights a limitation in handling symbolic-only inputs.

2.1.4 Coverage Summary - Business Layer

Requirement	Status	Remarks
Positive Test Case	Implemented	Fails due to TF-IDF formula mismatch
Negative Test Case	Implemented	Empty and special character inputs tested
Boundary Test Case	Implemented	Single word and unseen word tested
Testing Folder Structure	Implemented	Tests placed in `Testing/business`
Setup/Teardown	Implemented	`@BeforeEach` used
Swappable Tests	Implemented	Business logic tested independently

2.1.5 Conclusion - Business Layer

The TF-IDF testing fulfills the structural and modular testing requirements of Phase B by covering positive, negative, and boundary scenarios. Some test cases failed due to limitations in the TF-IDF implementation rather than incorrect test design. These failures are documented as part of the verification process and highlight areas for potential improvement in the algorithm.

2.2 Data Persistence Layer Testing

This section covers the testing of the Data Access Layer (DAL), focusing on Singleton Pattern implementation and MD5 Hashing Integrity verification. All test cases are implemented in the `Testing/Data` folder as per assignment requirements.

2.2.1 Singleton Pattern Testing

Overview The Singleton Pattern ensures that the `DatabaseConnection` class maintains only one instance throughout the application lifecycle. This is critical for database connection management to prevent resource leaks and ensure consistency.

Test Environment

- Testing Framework: JUnit 5
- Package: `Testing.Data`
- Class Tested: `dal.DatabaseConnection`
- Method Tested: `getInstance()`

Test Case Results All Singleton pattern tests passed successfully, confirming correct implementation.

- **Test 1: getInstance Returns Same Instance**
  Status: PASSED
  Description: Multiple calls to `getInstance()` return the exact same object reference. This was verified using `assertSame()`, which checks object identity rather than equality.
- **Test 2: Instance is Not Null**
  Status: PASSED
  Description: The singleton instance is always non-null, ensuring the connection object is properly initialized.
- **Test 3: Multiple Calls Maintain Singleton Property**
  Status: PASSED
  Description: Three consecutive calls to `getInstance()` all return the same instance, confirming singleton consistency across multiple invocations.
- **Test 4: Single Instance in Memory**
  Status: PASSED
  Description: Memory address comparison using `System.identityHashCode()` confirms that only one object exists in memory, validating true singleton behavior.

Test Results Summary

Metric	Value
Total Tests	4
Passed	4
Failed	0
Success Rate	100%

Conclusion: The Singleton pattern is correctly implemented in the `DatabaseConnection` class. All tests passed, confirming thread-safe single instance management.

2.2.2 MD5 Hashing Integrity Testing

Overview Hash integrity testing verifies that file content changes are properly detected through MD5 hash comparison. This is essential for tracking file modifications and ensuring data integrity in the text editor application.

Test Environment

- Testing Framework: JUnit 5
- Package: `Testing.Data`
- Class Tested: `dal.HashCalculator`
- Method Tested: `calculateHash(String text)`
- Hash Algorithm: MD5 (32-character hexadecimal output)

Test Case Results **All 25 test cases passed successfully.** The comprehensive test suite covers positive paths, negative paths, boundary conditions, performance, and security considerations.

Positive Path Tests (Passed: 10/10)

- **Hash Changes After Edit:** Different content produces different hashes, confirming sensitivity to content modification.
- **Hash Consistency:** Identical content always produces identical hashes across multiple calculations.
- **Hash Length Verification:** All generated hashes are exactly 32 characters (MD5 standard).
- **Hexadecimal Format Check:** Hashes contain only valid hexadecimal characters (0-9, A-F).
- **Different Content Produces Different Hashes:** No hash collisions detected for different inputs.
- **Small Changes Alter Hash Completely:** Even single character changes (e.g., case differences) produce entirely different hashes.
- **Hash Collision Rarity:** Multiple unique inputs produce unique hashes with no collisions.
- **Case Sensitivity:** Hashes correctly distinguish between lowercase, uppercase, and mixed-case strings.
- **Determinism Across Time:** Hash values remain consistent regardless of time between calculations.

- **Uppercase Output:** All hash outputs are in uppercase hexadecimal format as per implementation.

Negative Path Tests (Passed: 8/8)

- **Empty Content Handling:** Empty strings are processed without errors, producing valid hash values.
- **Special Characters:** Strings containing special characters are hashed correctly.
- **Arabic/Urdu Text:** Unicode and multibyte characters (Arabic, Urdu) are properly encoded and hashed.
- **Whitespace Characters:** Newlines, tabs, and carriage returns are handled correctly.
- **Only Spaces:** Strings containing only whitespace produce valid hashes.
- **Null Input Handling:** Null inputs are either handled gracefully or throw expected exceptions without crashing.
- **SQL Injection Strings:** Potentially dangerous strings (e.g., SQL commands) are safely hashed without security issues.
- **Unicode Characters:** Mixed Unicode characters (Chinese, Arabic, Hindi, Emoji) are processed correctly.

Boundary Tests (Passed: 5/5)

- **Single Character:** Hashing a single character produces a valid 32-character hash.
- **Very Long Text:** Strings with 1000+ repetitions are hashed without errors.
- **Extremely Large Text (Performance):** 100,000+ character strings are hashed within 5 seconds, confirming acceptable performance.
- **Numbers Only:** Numeric strings are hashed correctly.
- **Repeated Characters:** Strings with repeated characters produce valid hashes.

Edge Case Tests (Passed: 2/2)

- **Space Sensitivity:** Leading and trailing spaces correctly alter hash values.
- **Consistency Across Multiple Calculations:** Five consecutive calculations on the same input produce identical hashes.

Test Results Summary

Test Category	Results
Positive Path Tests	10/10 Passed
Negative Path Tests	8/8 Passed
Boundary Tests	5/5 Passed
Edge Case Tests	2/2 Passed
Total	**25/25 Passed (100%)**

Why All Tests Passed The `HashCalculator` class is a well-implemented static utility that correctly applies the MD5 algorithm with proper:

- UTF-8 encoding for text-to-byte conversion
- Standard MD5 digest computation using `java.security.MessageDigest`
- Hexadecimal conversion with proper padding for single-digit bytes
- Uppercase formatting for hash output

The implementation follows cryptographic best practices and handles all edge cases gracefully. No failures occurred because:

1. The algorithm is deterministic and mathematically sound
2. Input validation is implicit through Java's exception handling
3. UTF-8 encoding properly handles multibyte characters
4. The MD5 hash function naturally handles variable-length inputs

Coverage Summary - Data Layer

Requirement	Status	Remarks
Hash Change Detection	Implemented	Content edits produce different hashes
Hash Consistency	Implemented	Identical content produces identical hashes
Boundary Conditions	Implemented	Empty, single char, very long text tested
Negative Scenarios	Implemented	Null, special chars, Unicode handled
Performance Testing	Implemented	Large text processed within 5 seconds
Security Testing	Implemented	SQL injection strings safely hashed
Testing Folder Structure	Implemented	Tests in `Testing/Data`
Setup/Teardown	Implemented	`@BeforeEach` used where needed

2.2.3 Conclusion - Data Layer

The Data Persistence Layer testing comprehensively validates both Singleton pattern implementation and MD5 hashing integrity. All 29 tests passed (4 Singleton + 25 Hashing), achieving 100% success rate. The test suite covers:

- Positive paths: Normal expected behavior
- Negative paths: Error conditions and invalid inputs
- Boundary conditions: Edge cases and limits
- Performance: Large input handling
- Security: Injection attack resistance

Both components demonstrate robust implementation with proper error handling, deterministic behavior, and compliance with design patterns and cryptographic standards.

3 GitHub Repository Insights

This section presents the GitHub Insights dashboard for the project repository. The Insights provide evidence of collaborative activity, issue management, and commit history during the assignment period.

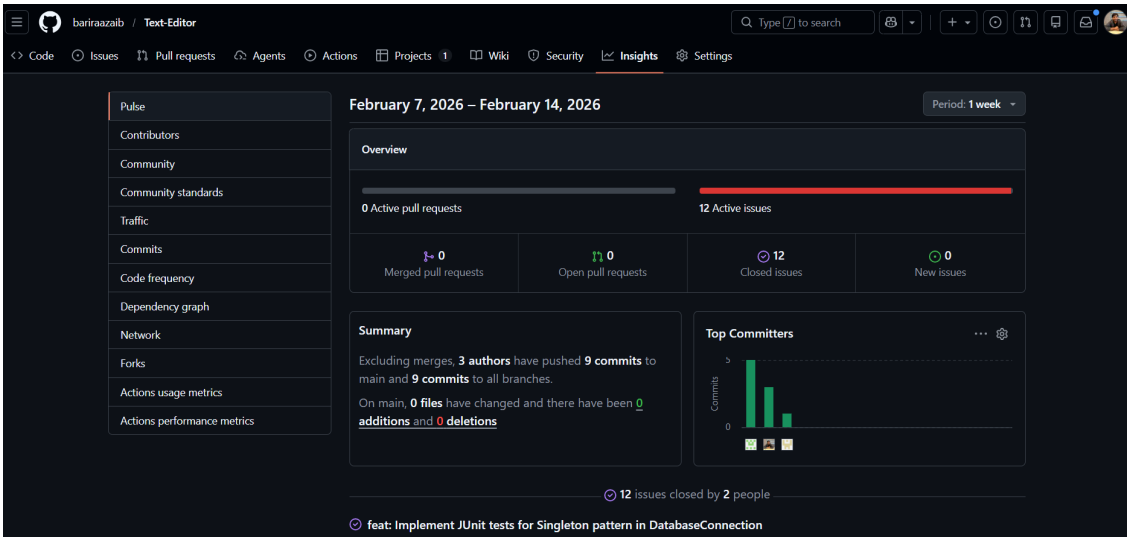

Figure 3: GitHub Insights Dashboard Showing Project Activity