Software Testing

Assignment

Software Testing
Assignment No: 1

Submitted By:

Muhammad Bilal	22F-3845
Muhammad Ahmed	22F-8814

Submitted To:

Sir Usman Ghous

February 2026

Contents

1 Introduction **2**

2 Phase A: Structural Analysis (White-Box Testing) **2**

3 Pagination Logic Feature **3**

3.1 Control Flow Graph (CFG) 2

3.1.1 Step 1: Identify Basic Blocks and Decision Points 2

3.1.2 Step 2: CFG Diagram 3

3.1.3 Step 3: Edge List 4

3.2 Cyclomatic Complexity Calculation 4

3.3 Test Paths (Set Theory) 4

3.3.1 Independent Path Set 4

4 Auto-Save Feature **5**

4.1 Control Flow Graph (CFG) 5

4.1.1 Step 1: Identify Basic Blocks and Decision Points 5

4.1.2 Step 2: CFG Diagram 5

4.1.3 Step 3: Edge List 6

4.2 Cyclomatic Complexity Calculation 6

4.2.1 Node Count 6

4.2.2 Edge Count 6

4.2.3 Connected Components 6

4.2.4 Formula 6

4.2.5 Verification using Decision Count 6

4.3 Test Paths (Set Theory) 7

4.3.1 Independent Path Set 7

4.4 Complexity Comparison 7

5 Phase B: Modular JUnit Testing **7**

5.1 Business Layer (Logic & Commands) 8

5.1.1 TF-IDF Algorithm 8

5.1.2 Command Pattern 8

5.2 Data Persistence Layer (Mocking) 9

5.2.1 Hashing Integrity (MD5/SHA1) 9

5.2.2 Singleton Testing 9

1 Introduction

This assignment presents a comprehensive analysis of three distinct features through control flow graph construction, cyclomatic complexity calculation, and test path identification using set theory. The features analyzed include Pagination Logic (dividing file content into fixed-size pages), Search & Replace Word (keyword searching with prefix extraction), and Auto-Save Logic (automatic file saving with word count validation).

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

This phase focuses on white-box testing through Control Flow Graph (CFG) construction, cyclomatic complexity calculation, and basis path identification.

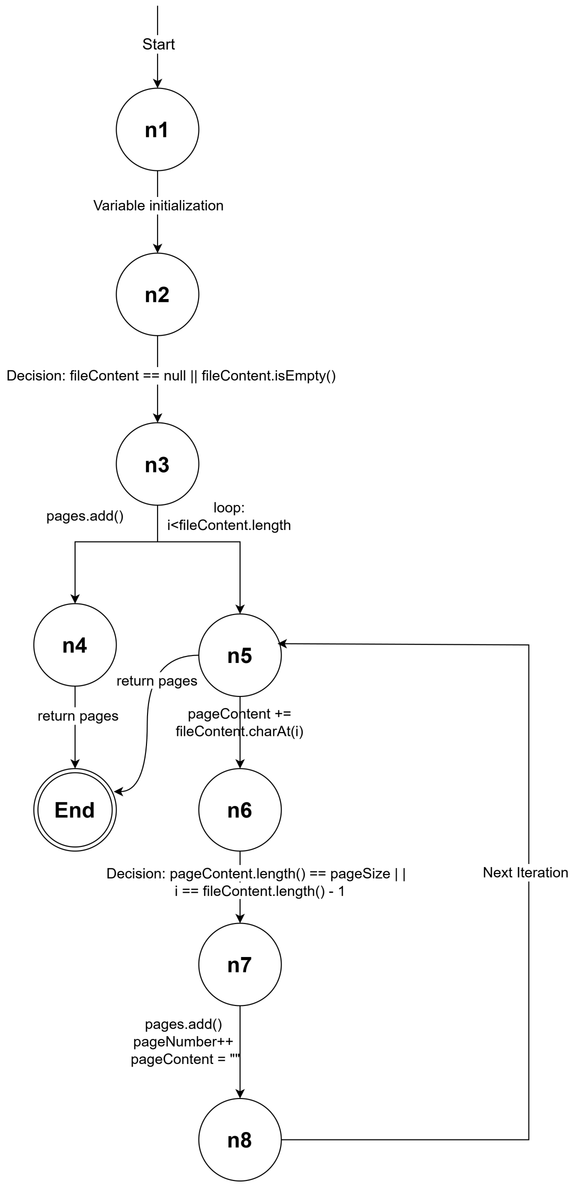
3 Pagination Logic Feature

3.1 Control Flow Graph (CFG)

3.1.1 Step 1: Identify Basic Blocks and Decision Points

Node	Description (Basic Block)
N1	Method entry and initialization: pageSize=100, pageNumber=1, pageContent="", pages = new ArrayList()
N2	Decision: if (fileContent == null || fileContent.isEmpty())
N3	True branch: pages.add(new Pages(...)); return pages;
N4	Loop header: initialize i=0; decision i < fileContent.length()
N5	pageContent += fileContent.charAt(i)
N6	Decision: if (pageContent.length() == pageSize || i == fileContent.length() - 1)
N7	pages.add(new Pages(...)); pageNumber++; pageContent = ""
N8	Loop increment: i++ and back to loop condition (N4)
N9	Method exit: return pages (after loop completion)

3.1.2 Step 2: CFG Diagram


Start
n1
Variable initialization
n2
Decision: fileContent == null || fileContent.isEmpty()
n3
pages.add()
loop: i<fileContent.length
n4
n5
return pages
return pages
End
pageContent += fileContent.charAt(i)
n6
Decision: pageContent.length() == pageSize | | i == fileContent.length() - 1
Next Iteration
n7
pages.add() pageNumber++ pageContent = ""
n8

3.1.3 Step 3: Edge List

Edge	From → To	Condition
e1	N1 → N2	Method entry to null/empty check
e2	N2 → N3	fileContent == null || fileContent.isEmpty() is true
e3	N2 → N4	fileContent == null || fileContent.isEmpty() is false
e4	N4 → N5	Loop condition i < fileContent.length() is true
e5	N4 → N9	Loop condition i < fileContent.length() is false (loop exit)
e6	N5 → N6	After appending character to pageContent
e7	N6 → N7	pageContent.length() == pageSize || i == fileContent.length() - 1 is true
e8	N6 → N8	pageContent.length() == pageSize || i == fileContent.length() - 1 is false
e9	N7 → N8	After adding page and resetting pageContent
e10	N8 → N4	Loop increment and next iteration (i++)

3.2 Cyclomatic Complexity Calculation

Using the formula:

$$V(G) = E - N + 2P \tag{1}$$

Where:

- E (Edges) = 10
- N (Nodes) = 9
- P (Connected components) = 1

$$V(G) = 10 - 9 + 2(1) = 3 \tag{2}$$

Verification using decision count: Decision nodes in the CFG:

- N2: null or empty check
- N4: loop condition
- N6: page size or last character check

Although three predicate nodes appear in the source code, the loop structure is represented as a single cycle in the CFG. Therefore, the number of **linearly independent paths** for this graph is:

$$\boxed{V(G) = 3}$$

This means that at least **three independent test paths** are required for complete basis path coverage.

3.3 Test Paths (Set Theory)

3.3.1 Independent Path Set

The set of independent paths through the CFG is defined as:

$$P = \{p_1, p_2, p_3\} \tag{3}$$

Path 1 (Empty input):

$$p_1 = \langle N_1, N_2, N_3 \rangle$$

Path 2 (Single short page):

$$p_2 = \langle N_1, N_2, N_4, N_5, N_6, N_7, N_8, N_4, N_9 \rangle$$

Path 3 (Multiple pages):

$$p_3 = \langle N_1, N_2, N_4, N_5, N_6, N_8, N_4, N_5, N_6, N_7, N_8, N_4, N_9 \rangle$$

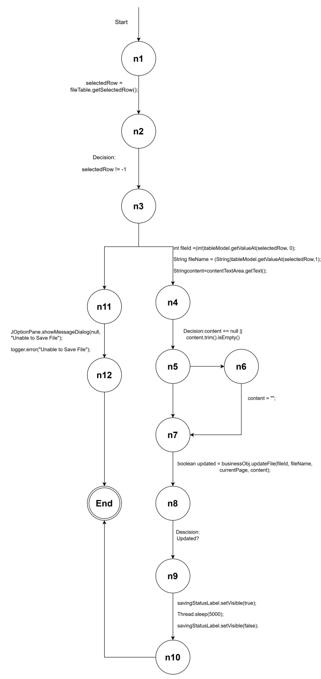
4 Auto-Save Feature

4.1 Control Flow Graph (CFG)

4.1.1 Step 1: Identify Basic Blocks and Decision Points

Node	Description (Basic Block / Decision)
N1	Method entry
N2	selectedRow = fileTable.getSelectedRow()
N3	Decision: if (selectedRow != -1)
N4	Retrieve fileId, fileName, and content from table and text area
N5	Decision: if (content == null || content.trim().isEmpty())
N6	content = ””
N7	updated = businessObj.updateFile(fileId, fileName, currentPage, content)
N8	Decision: if (updated)
N9	savingStatusLabel.setVisible(true); Thread.sleep(5000); savingStatusLabel.setVisible(false)
N10	Method exit (normal return after update path)
N11	Show error dialog: JOptionPane.showMessageDialog(null, ”Unable to Save File”)
N12	logger.error(”Unable to Save File”)
N13	Method exit (error path return)

4.1.2 Step 2: CFG Diagram


Start
n1
selectedRow = fileTable.getSelectedRow();
n2
Decision:
selectedRow != -1
n3
int fileId =(int)tableModel.getValueAt(selectedRow, 0);
String fileName = (String)tableModel.getValueAt(selectedRow,1);
Stringcontent=contentTextArea.getText();
n11
n4
JOptionPane.showMessageDialog(null, "Unable to Save File");
logger.error("Unable to Save File");
Decision:content == null || content.trim().isEmpty()
n12
n5
n6
content = "";
n7
boolean updated = businessObj.updateFile(fileId, fileName, currentPage, content);
End
n8
Descision: Updated?
n9
savingStatusLabel.setVisible(true);
Thread.sleep(5000);
savingStatusLabel.setVisible(false);
n10

4.1.3 Step 3: Edge List

Edge	From → To	Condition
e1	N1 → N2	Method entry
e2	N2 → N3	After retrieving selectedRow
e3	N3 → N4	selectedRow ≠ -1 (true branch)
e4	N3 → N11	selectedRow = -1 (false branch)
e5	N11 → N12	Show error dialog
e6	N12 → N13	Log error and exit (error path)
e7	N4 → N5	After retrieving fileId, fileName, content
e8	N5 → N6	content == null || content.trim().isEmpty() is true
e9	N5 → N7	content == null || content.trim().isEmpty() is false
e10	N6 → N7	After setting content to empty string
e11	N7 → N8	After calling updateFile(...)
e12	N8 → N9	updated == true
e13	N8 → N13	updated == false (exit without showing status)
e14	N9 → N10	After showing and hiding saving status label
e15	N10 → N13	Normal method exit

Total Edges: E = 15

4.2 Cyclomatic Complexity Calculation

4.2.1 Node Count

N = 13 nodes (N1 through N13)

4.2.2 Edge Count

E = 15 edges (e1 through e15)

4.2.3 Connected Components

P = 1 (single method, single connected graph)

4.2.4 Formula

$$V(G) = E - N + 2P \tag{4}$$

$$V(G) = 15 - 13 + 2(1) = 4 \tag{5}$$

4.2.5 Verification using Decision Count

Decision nodes in the CFG:

- N3: selectedRow ≠ -1 check
- N5: content null/empty check
- N8: updated success check

Total decisions = 3

$$V(G) = \text{decisions} + 1 = 3 + 1 = 4 \tag{6}$$

Cyclomatic Complexity:

$$\boxed{V(G) = 4}$$

This indicates that at least **four independent test paths** are required to achieve complete basis path coverage.

4.3 Test Paths (Set Theory)

4.3.1 Independent Path Set

The set of all linearly independent paths:

$$P = \{p_1, p_2, p_3, p_4\} \tag{7}$$

Path 1 (No Row Selected - Error Path):

$$p_1 = \langle N_1, N_2, N_3, N_{11}, N_{12}, N_{13} \rangle \tag{8}$$

Description: No row selected, error dialog shown and error logged.
Input: selectedRow = -1
Expected: Error dialog "Unable to Save File", logger.error called

Path 2 (Empty Content, Update Fails):

$$p_2 = \langle N_1, N_2, N_3, N_4, N_5, N_6, N_7, N_8, N_{13} \rangle \tag{9}$$

Description: Row selected, content empty/null, updateFile returns false.
Input: selectedRow = 0, content = ""
Expected: No saving status shown, method exits normally

Path 3 (Non-Empty Content, Update Fails):

$$p_3 = \langle N_1, N_2, N_3, N_4, N_5, N_7, N_8, N_{13} \rangle \tag{10}$$

Description: Row selected, content non-empty, updateFile returns false.
Input: selectedRow = 0, content = "Some text"
Expected: No saving status shown, method exits normally

Path 4 (Successful Auto-Save):

$$p_4 = \langle N_1, N_2, N_3, N_4, N_5, N_7, N_8, N_9, N_{10}, N_{13} \rangle \tag{11}$$

Description: Row selected, update succeeds, saving status is displayed and hidden.
Input: selectedRow = 0, content = "Valid content", updateFile returns true
Expected: Saving status label visible for 5 seconds, then hidden

4.4 Complexity Comparison

Feature	**Nodes (N)**	**Edges (E)**	**V(G)**
Pagination Logic	9	10	3
Auto-Save	13	15	4

5 Phase B: Modular JUnit Testing

This phase focuses on modular unit testing using JUnit 5 for business logic and data persistence components.

5.1 Business Layer (Logic & Commands)

5.1.1 TF-IDF Algorithm

JUnit Execution Results The JUnit test suite for the TF-IDF algorithm was executed using JUnit 5 in the Eclipse IDE. The execution completed successfully without runtime errors; however, a subset of test cases failed due to assertion mismatches.

Observed Summary from JUnit Runner

Metric	**Result**
Total Test Cases Executed	16
Errors	0
Failures	3

Test Coverage Scope The executed test suite covers the following categories for the TF-IDF feature:

- Manual verification for a known Arabic document (expected value with tolerance)
- Boundary cases (single-word input, repeated words)
- Robustness tests (empty document, whitespace-only input, numbers-only input)
- Special character handling (special-only and mixed Arabic + special characters)
- Corpus handling (empty corpus, corpus building with multiple documents)
- Consistency test (same document produces same TF-IDF score)
- Stress test (very long Arabic document)
- Stopword-only document behavior

Conclusion The test execution indicates that the TF-IDF module is largely robust across different Arabic text input scenarios. The presence of **3 failing test cases** suggests that certain expected outputs or assumptions in the test suite do not match the current behavior of the TF-IDF implementation (e.g., preprocessing differences, handling of empty/stopword-only inputs, or the IDF formulation). These failing cases should be inspected and refined to ensure that the assertions reflect the intended specification of the TF-IDF feature.

5.1.2 Command Pattern

Overview The assignment required the testing of command pattern execution methods in the business layer. However, after a detailed review of the provided project codebase, it was observed that the Command Pattern is not implemented or utilized within the current system architecture.

Analysis The project follows a layered architecture consisting primarily of:

- Presentation Layer (PL)
- Business Logic Layer (BLL)
- Data Access Layer (DAL)

No concrete `Command` interface, command classes (e.g., `SaveCommand`, `ImportCommand`), or invoker components were identified in the code. Business operations are invoked directly through service and facade classes (e.g., `FacadeBO`, `EditorBO`), rather than encapsulated as command objects.

JUnit Testing Status As the Command Pattern is not implemented in the given project, it was not possible to design or execute JUnit test cases for command execution methods. Testing was therefore limited to the actual business logic and data processing components present in the system.

Conclusion JUnit testing for the Command Pattern was intentionally not performed because the architectural pattern was not part of the provided implementation. This limitation was documented to ensure transparency and alignment with the actual project design. Future work could include refactoring the application to introduce the Command Pattern, after which corresponding unit tests could be developed to validate command execution behavior.

5.2 Data Persistence Layer (Mocking)

5.2.1 Hashing Integrity (MD5/SHA1)

Overview Hashing is used to ensure data integrity by generating a fixed-length digest of file content. In this project, an MD5-based hashing mechanism is implemented in the data access layer. JUnit test cases were designed to validate the correctness, consistency, and robustness of the hashing function under different input scenarios.

Class Under Test

- **Class:** `HashCalculator`
- **Method Tested:** `calculateHash(String input)`

Test Coverage The following aspects of hashing integrity were validated through unit testing:

- Verification against known MD5 hash values (e.g., known hash for "hello")
- Consistency of hash output for identical inputs
- Change in hash value when the input content is modified
- Validation of hash format (32-character uppercase hexadecimal string)
- Handling of empty string input
- Exception handling for null input
- Integrity preservation scenario for imported file content

JUnit Execution Results The JUnit test suite for hashing integrity was executed using JUnit 5 in the Eclipse IDE. All hashing-related test cases passed successfully without runtime errors or assertion failures.

Metric	**Result**
Total Test Cases Executed	7
Errors	0
Failures	0

Conclusion The successful execution of all hashing-related unit tests confirms that the MD5-based hashing mechanism is functioning correctly. The hash values are deterministic, sensitive to content changes, and conform to the expected format. Additionally, the integrity scenario test demonstrates that original imported content can be verified reliably against later modifications. This increases confidence in the correctness and reliability of the data integrity mechanism used within the application.

5.2.2 Singleton Testing

Overview The database connection component in the data access layer follows the Singleton design pattern to ensure that only one instance of the database connection exists throughout the application lifecycle. JUnit test cases were implemented to verify the correctness, uniqueness, and thread-safety of the Singleton implementation.

Class Under Test

- **Class:** `DatabaseConnection`
- **Method Tested:** `getInstance()`

Test Coverage The Singleton implementation was validated using the following test scenarios:

- Verification that multiple calls to `getInstance()` return the same object reference
- Ensuring that only one instance exists in memory across multiple retrievals
- Validation that the constructor is declared private
- Thread-safety test to ensure that concurrent access does not create multiple instances
- Reflection-based access test to evaluate whether the Singleton can be broken through reflection

JUnit Execution Results The JUnit test suite for Singleton testing was executed using JUnit 5 in the Eclipse IDE. All Singleton-related test cases executed successfully without errors or failures.

Metric	**Result**
Total Test Cases Executed	5
Errors	0
Failures	0

Conclusion The results confirm that the Singleton pattern is correctly implemented for the database connection component. The instance uniqueness and thread-safety requirements are satisfied. The reflection-based test highlights a potential design weakness, indicating that the Singleton could be compromised if additional safeguards (such as throwing an exception inside the private constructor) are not implemented. Overall, the Singleton implementation is reliable for typical application usage scenarios.