# Movie Recommendation System

Software Testing

December 17, 2025

## Table of Contents


**1 1. Project Overview** **2**

**2 2. Integration Testing** **2**
2.1 2.1 Test Classes and Methodology . . . . . 2
2.1.1 2.1.1 DataStoreIntegrationTest . . . . . 2
2.1.2 2.1.2 MainIntegrationTest . . . . . 3
2.1.3 2.1.3 RecommenderIntegrationTest . . . . . 4
2.2 2.2 Integration Testing Summary . . . . . 4

**3 3. White Box Testing** **5**
3.1 3.1 Coverage-Based Testing . . . . . 5
3.1.1 3.1.1 Coverage Analysis . . . . . 5
3.1.2 3.1.2 Test Cases for Path Coverage . . . . . 7
3.1.3 3.1.3 Coverage Metrics Achieved . . . . . 8
3.2 3.2 Data Flow Testing . . . . . 8
3.2.1 3.2.1 MovieParser.parseMovies() Data Flow Testing . . . . . 9
3.2.2 3.2.2 UserParser.parseUsers() Data Flow Testing . . . . . 11
3.2.3 3.2.3 Recommender.recommendMovies() Data Flow Testing . . . 14
3.2.4 3.2.4 Additional Data Flow Tests . . . . . 17
3.2.5 3.2.5 Data Flow Testing Summary . . . . . 17

**4 4. Overall Testing Summary** **18**
4.1 4.1 Test Distribution . . . . . 18
4.2 4.2 Testing Techniques Applied . . . . . 18
4.2.1 **Integration Testing (Top-Down)** . . . . . 18
4.2.2 **White-Box Testing: Coverage-Based** . . . . . 18
4.2.3 **White-Box Testing: Data Flow** . . . . . 18
4.3 4.3 Key Strengths . . . . . 18
4.4 4.4 Testing Tools . . . . . 19

**5 5. Conclusion** **19**

# 1 1. Project Overview

The **Movie Recommendation System** is a Java-based application that generates personalized movie recommendations for users based on genre matching. The system:

- Parses movie and user data from text files
- Validates data integrity (referential integrity, format constraints)
- Generates recommendations by matching genres from users' liked movies
- Writes recommendations to an output file

The system follows a modular architecture with clear separation of concerns: parsing, data storage, recommendation logic, and output writing.

---

# 2 2. Integration Testing

Integration testing was performed using the **top-down approach**, where higher-level components are tested first with lower-level dependencies either real or mocked. Tests are located in `src/test/java/com/example/top_down/`.

## 2.1 2.1 Test Classes and Methodology

### 2.1.1 2.1.1 DataStoreIntegrationTest

**Purpose:** Verify integration between the `DataStore` class and domain entities (`Movie`, `User`).

**Test Cases:**

#### 2.1.1.1 TC-INT-1: `testAddMovieAndUser_Integration()`

- **Objective:** Validate that DataStore correctly accepts and stores valid movies and users
- **Setup:**
  - Create 2 movies: "The Matrix" (M001) and "Inception" (I002)
  - Create 1 user: "Alice" (123456789) who likes "M001"
- **Actions:**
  - Add movies to DataStore via `addMovie()`
  - Add user to DataStore via `addUser()`
- **Assertions:**
  - Verify `getMovies()` returns exactly 2 movies
  - Verify `getUsers()` returns exactly 1 user
  - Verify `getMovieById("M001")` retrieves correct movie
- **Result:** Confirms successful integration of storage operations

#### 2.1.1.2 TC-INT-2: `testAddUser_InvalidMovieReference_ThrowsException()`

- **Objective:** Verify integrity validation catches invalid movie references
- **Setup:**

  - Create 1 movie: "The Matrix" (M001)
  - Create 1 user referencing non-existent movie "INVALID_ID"
- **Actions:**
  - Attempt to add user with invalid reference
- **Assertions:**
  - Expect `DataIntegrityException` to be thrown
  - Verify exception message contains "movie with ID INVALID_ID does not exist"
- **Result:** Confirms referential integrity enforcement

#### 2.1.1.3 TC-INT-3: `testAddMovie_DuplicateNumericId_ThrowsException()`

- **Objective:** Verify duplicate numeric ID detection across different prefixes
- **Setup:**
  - Add movie with ID "TM001" (numeric part: 001)
  - Attempt to add movie with ID "JW001" (same numeric part: 001)
- **Actions:**
  - Add first movie successfully
  - Attempt to add second movie with duplicate numeric portion
- **Assertions:**
  - Expect `DataIntegrityException` to be thrown
  - Verify duplicate detection works regardless of prefix
- **Result:** Confirms business rule enforcement for unique numeric IDs

---

### 2.1.2 2.1.2 MainIntegrationTest

**Purpose:** End-to-end integration test of the entire workflow using the top-down approach with mocked dependencies.

**Testing Strategy:** Uses **Mockito spies** to create a partially mocked `Main` instance where factory methods return mocks instead of real objects.

#### 2.1.2.1 TC-INT-4: `testMain_Integration_WithMocks()`

- **Objective:** Verify correct orchestration of all components in the main workflow
- **Setup:**
  - Mock `Parser`, `ParseResult`, `DataStore`, `Recommender`, `RecommendationWriter`
  - Configure mocks:
    - Parser returns ParseResult with test movie and user
    - DataStore returns list with test user
    - Recommender returns empty recommendation list
- **Actions:**
  - Create spy of Main class
  - Override factory methods to return mocks:
    - `createParser()` → mock parser
    - `createDataStore()` → mock data store
    - `createRecommender()` → mock recommender
    - `createRecommendationWriter()` → mock writer

  - Execute `main()` method
- **Verifications:**
  - `parser.parse(moviesPath, usersPath)` called exactly once
  - `dataStore.getUsers()` called exactly once
  - `recommender.recommendMovies(user)` called exactly once
  - `writer.writeRecommendations(anyMap(), eq(outputPath))` called exactly once
- **Result:** Confirms correct component interaction and workflow orchestration

**Key Integration Pattern:** The spy pattern allows testing the real orchestration logic in `Main.main()` while controlling dependencies through method overrides.

---

#### 2.1.3 2.1.3 RecommenderIntegrationTest

**Purpose:** Verify integration between `Recommender` and `DataStore` with real recommendation algorithm logic.

##### 2.1.3.1 TC-INT-5: `testRecommendMovies_Integration()`

- **Objective:** Validate recommendation algorithm with mocked data store
- **Setup:**
  - Mock `DataStore` with 3 movies:
    - "The Matrix" (M001): Action, Sci-Fi, Thriller
    - "Inception" (I002): Action, Sci-Fi, Mystery
    - "The Shawshank Redemption" (S003): Drama
  - Create user "Alice" who liked "M001"
  - Configure DataStore mock:
    - `getMovieById("M001")` returns The Matrix
    - `getMovies()` returns all 3 movies
- **Actions:**
  - Execute `recommender.recommendMovies(user)`
- **Assertions:**
  - Result list size equals 1
  - Recommended movie is "Inception"
  - Excluded "The Matrix" (already liked)
  - Excluded "The Shawshank Redemption" (no genre match)
- **Verifications:**
  - `dataStore.getMovieById()` called once per liked movie
  - `dataStore.getMovies()` called once
- **Result:** Confirms correct integration between recommender and data store

**Algorithm Validation:** 1. Extracts genres from liked movies: [Action, Sci-Fi, Thriller] 2. Filters candidates by genre match 3. Excludes already-liked movies 4. Returns matching movies

---

### 2.2 2.2 Integration Testing Summary

| Test Class | Test Cases | Components Tested | Key Validations |
|---|---|---|---|
| DataStoreIntegrationTest | 3 | DataStore Movie/User | Storage, integrity checks |
| MainIntegrationTest | 1 | All components via Main | Workflow orchestration |
| RecommenderIntegrationTest | 1 | Recommender DataStore | Algorithm correctness |

**Total Integration Tests:** 5 test methods across 3 test classes

**Integration Strategy Benefits:** - Top-down approach allows early testing of critical workflows - Strategic mocking isolates integration points - Verification of method calls confirms correct component interaction - Real business logic validated with controlled test data

---

# 3 3. White Box Testing

White box testing examines the internal structure and logic of the code. This project implements two complementary white-box techniques:

1. **Coverage-Based Testing** (statement, branch, condition coverage)
2. **Data Flow Testing** (definition-use paths)

---

## 3.1 3.1 Coverage-Based Testing

**Location:** `src/test/java/com/example/wbt/path_coverage/`

**Target Class:** `DataStore` (chosen for its complex control flow with loops and conditional validation)

**Coverage Criteria:** - **Statement Coverage:** Every executable statement executed at least once - **Branch Coverage:** Every decision branch (true/false) taken at least once - **Condition Coverage:** Each boolean sub-expression evaluated to both true and false

---

### 3.1.1 3.1.1 Coverage Analysis

The `DataStore` class contains four methods with distinct control flow patterns:

#### 3.1.1.1 Method 1: `addMovie(Movie movie)` Control Flow:

```
START
  → Extract numeric part of movie ID
  → FOR each existing movie:
      IF numeric parts match:
        → THROW DataIntegrityException
  → Add movie to list
END
```

**Paths Identified:** - **Path 1:** Empty list (loop executes 0 times) - **Path 2:** Non-empty list, no duplicates found (loop executes N times, condition always false) - **Path 3:** Duplicate detected (loop executes, condition becomes true)

---

#### 3.1.1.2 Method 2: `addUser(User user)` Control Flow:

```
START
  → Call checkIntegrity(user)
      IF integrity check passes:
        → Add user to list
      IF integrity check fails:
        → THROW DataIntegrityException
END
```

**Paths Identified:** - **Path 4:** Valid user (checkIntegrity returns true) - **Path 5:** Invalid user - missing movie reference (checkIntegrity throws exception) - **Path 6:** Invalid user - duplicate user ID (checkIntegrity throws exception)

---

#### 3.1.1.3 Method 3: `checkIntegrity(User user)` (private) Control Flow:

```
START
  → FOR each liked movie ID:
      IF movie doesn't exist:
        → THROW DataIntegrityException
  → FOR each existing user:
      IF user ID matches:
        → THROW DataIntegrityException
  → RETURN true
END
```

**Note:** This method never returns false; it either returns true or throws an exception.

**Paths Identified:** - Liked movie exists (continue iteration) - Liked movie doesn't exist (throw exception) - Duplicate user ID (throw exception) - All checks pass (return true)

---

#### 3.1.1.4 Method 4: `getMovieById(String id)` Control Flow:

```
START
  → FOR each movie:
      IF movie.id equals id:
        → RETURN movie
  → RETURN null
END
```

**Paths Identified:** - **Path 7:** Movie found (return early) - **Path 8:** Movie not found (complete loop, return null)

---

### 3.1.2 3.1.2 Test Cases for Path Coverage

**Test Class:** `DataStoreClassTest`

**3.1.2.1 Testing addMovie() - Paths 1-3 TC-COV-1: `addMovieFirst()` - Path Covered:** Path 1 (loop 0 iterations) - **Setup:** - Empty DataStore - Single movie: "Matrix" (M001) - **Action:** `store.addMovie(m1)` - **Assertion:** `getMovies().size() == 1` - **Coverage:** - Statement: All statements in method executed - Branch: Loop condition false immediately - Condition: `i < movies.size()` evaluates to false

**TC-COV-2: `addMovieUnique()` - Path Covered:** Path 2 (loop with no duplicates) - **Setup:** - Add "Matrix" (M001) - numeric part: 001 - Add "Inception" (I002) - numeric part: 002 - **Action:** Add both movies sequentially - **Assertion:** `getMovies().size() == 2` - **Coverage:** - Statement: Loop body executed, condition never satisfied - Branch: Loop iterates (true), duplicate check (false) - Condition: `i < movies.size()` true, `numericPart.equals()` false

**TC-COV-3: `addMovieDuplicateNumeric()` - Path Covered:** Path 3 (duplicate detection) - **Setup:** - Add "Matrix" (M001) - numeric part: 001 - Attempt "Avatar" (A001) - numeric part: 001 - **Action:** `store.addMovie(duplicate)` - **Assertion:** Throws `DataIntegrityException` - **Coverage:** - Statement: Exception throw statement executed - Branch: Duplicate condition true - Condition: `numericPart.equals(otherNumericPart)` evaluates to true

---

**3.1.2.2 Testing addUser() - Paths 4-6 TC-COV-4: `addUserValid()` - Path Covered:** Path 4 (integrity check passes) - **Setup:** - DataStore with "Matrix" (M001) - User "Alice" (123456789) likes ["M001"] - **Action:** `store.addUser(u)` - **Assertion:** `getUsers().size() == 1` - **Coverage:** - Statement: All statements in addUser and checkIntegrity - Branch: checkIntegrity returns true, user added - Condition: Movie exists check true, no duplicate ID check true

**TC-COV-5: `addUserInvalid()` - Path Covered:** Path 5 (missing movie reference) - **Setup:** - Empty DataStore (no movies) - User "Alice" likes ["M001"] (doesn't exist) - **Action:** `store.addUser(u)` - **Assertion:** Throws `DataIntegrityException` with message "movie with ID M001 does not exist" - **Coverage:** - Statement: Exception path in checkIntegrity - Branch: Movie exists check fails - Condition: `getMovieById(id) == null` evaluates to true

**TC-COV-6: `addUserDuplicateId()` - Path Covered:** Path 6 (duplicate user ID) - **Setup:** - DataStore with "Matrix" (M001) - User "Alice" (123456789) already added - Attempt to add "Bob" (123456789) - same ID - **Action:** `store.addUser(u2)` - **Assertion:** Throws `DataIntegrityException` with message about duplicate ID - **Coverage:** - Statement: Duplicate ID check in checkIntegrity - Branch: Duplicate ID condition true - Condition: `existingUser.getId().equals(user.getId())` evaluates to true

---

#### 3.1.2.3 Testing getMovieById() - Paths 7-8 TC-COV-7: `getMovieByIdExists()`

- **Path Covered:** Path 7 (movie found) - **Setup:** - DataStore with "Matrix" (M001) - **Action:** `store.getMovieById("M001")` - **Assertion:** `assertNotNull(result)` and result is "Matrix" - **Coverage:** - Statement: Return statement inside loop - Branch: ID match condition true, early return - Condition: `movie.getId().equals(id)` evaluates to true

**TC-COV-8: `getMovieByIdNotFound()` - Path Covered:** Path 8 (not found) - **Setup:** - Empty DataStore - **Action:** `store.getMovieById("X999")` - **Assertion:** `assertNull(result)` - **Coverage:** - Statement: Return null statement after loop - Branch: Loop completes without match - Condition: All `movie.getId().equals(id)` evaluate to false

---

### 3.1.3 3.1.3 Coverage Metrics Achieved

**Coverage Report:** `src/test/java/com/example/wbt/path_coverage/DataStoreClassTest`

| Method | Statement Coverage | Branch Coverage | Condition Coverage |
|---|---|---|---|
| addMovie() | 100% | 100% | 100% |
| addUser() | 100% | 100% | N/A* |
| checkIntegrity() | 100% | 100% | 100% |
| getMovieById() | 100% | 100% | 100% |

**Overall Result:** 100% statement, branch, and condition coverage achieved

**Note:** *Condition coverage for `addUser()` is marked N/A because `checkIntegrity()` never returns false—it either returns true or throws an exception, so there's no true/false branch in the traditional sense.

---

## 3.2 3.2 Data Flow Testing

**Location:** `src/test/java/com/example/wbt/data_flow/`
**Documentation:** `docs/data_flow/`

Data flow testing focuses on **definition-use (DU) paths** for variables. A DU path is a path from where a variable is defined to where it's used, with no redefinition in between.

**Coverage Criteria:** - **All-Defs:** Every variable definition reaches at least one use - **All-Uses:** Every use of every definition is exercised - **All-DU-Paths:** Every definition-clear path from definition to use is tested

**Target Methods:** Three most complex methods identified through cyclomatic complexity and control flow analysis: 1. `MovieParser.parseMovies(String)` 2. `UserParser.parseUsers(String)` 3. `Recommender.recommendMovies(User)`

---

### 3.2.1 3.2.1 MovieParser.parseMovies() Data Flow Testing

**Method Signature:** `List<Movie> parseMovies(String moviesFileData)`

**Complexity Factors:** - While loop with manual index management (`i` incremented at multiple points) - Multiple guard/exception branches - Nested iteration for genre normalization - Variable `i` has multiple redefinitions

**Reference:** `docs/data_flow/movieparser-parseMovies.md`

---

#### 3.2.1.1 Variables Tracked

| Variable | Definition Points | Use Points |
|---|---|---|
| `lines` | `split("\n")` | Loop condition, array access |
| `i` | Initial `= 0`, three `i++` locations | Loop condition (P-use), array indexing (C-use) |
| `titleAndId` | `line.split(",", 2)` | Length check (P-use), array access (C-use) |
| `title` | `titleAndId[0].trim()` | Movie constructor (C-use) |
| `id` | `titleAndId[1].trim()` | Movie constructor, exception messages (C-use) |
| `line` | Two definitions: header line, genres line | Split operations (C-use) |
| `genresArray` | `line.split(",")` | Length check (P-use), digit regex (P-use), iteration (C-use) |

| Variable | Definition Points | Use Points |
|---|---|---|
| `genres` | `new ArrayList<>()` | add() calls (C-use), Movie constructor (C-use) |
| `movie` | `new Movie(...)` | List add (C-use) |

---

**3.2.1.2 Definition-Use Paths** **DU-Path 1:** `i` definition (initial) → P-use in `while (i < lines.length)`
**DU-Path 2:** `i++` (after header) → P-use in `if (i >= lines.length)`
**DU-Path 3:** `i` → C-use in `lines[i]` (header access)
**DU-Path 4:** `i++` (after header) → C-use in `lines[i]` (genres access)
**DU-Path 5:** `titleAndId` definition → P-use in `if (titleAndId.length < 2)`
**DU-Path 6:** `titleAndId` definition → C-use in `titleAndId[0]`, `titleAndId[1]`
**DU-Path 7:** `genresArray` → P-use in `if (genresArray.length > 1)`
**DU-Path 8:** `genresArray` → P-use in digit regex match
**DU-Path 9:** `genresArray` → C-use in for-each loop
**DU-Path 10:** `i++` (end of iteration) → P-use in next iteration's `while` condition

---

**3.2.1.3 Test Cases** **Test Class:** `MovieParserDataFlowTest`

**TC-DF-MP-1: `parseMovies_puseTitleAndIdLength_true_skipsLine_thenParsesNext()`** - **DU-Path:** titleAndId.length < 2 → true → i++ → continue → next iteration - **Input:** `BadLineWithoutComma` `The Matrix,TM001` `Action,SciFi` - **Expected:** Skips first line, parses "The Matrix" successfully - **Variables Covered:** `titleAndId` (P-use length check), `i` (redefinition and reuse)

**TC-DF-MP-2: `parseMovies_puseMissingGenres_true_throws()`** - **DU-Path:** i++ after header → i >= lines.length → true → throw exception - **Input:** `The Matrix,TM001` - **Expected:** Throws `MovieException` "Genres are missing for movie TM001" - **Variables Covered:** `i` (definition → P-use), `id` (C-use in exception message)

**TC-DF-MP-3: `parseMovies_puseInvalidGenres_secondTokenHasDigit_throws()`** - **DU-Path:** genresArray definition → P-use in digit regex → true → throw - **Input:** `The Matrix,TM001` `Action,Sc1Fi` - **Expected:** Throws `MovieException` "Genres are invalid for movie TM001" - **Variables Covered:** `genresArray` (P-use in regex), `id` (C-use in message)

**TC-DF-MP-4: `parseMovies_oneCompleteMovie_singleIteration()`** - **DU-Path:** i initial definition → one iteration → i++ → exit condition - **Input:** `Matrix,M001` `Action` - **Expected:** Returns list with 1 movie - **Variables Covered:** `i` (definition → multiple C-uses and P-uses in single iteration)

**TC-DF-MP-5: `parseMovies_multipleSkipLines_thenValidMovie_exercisesMultipleIF`** - **DU-Path:** Multiple i redefinitions with skip branches - **Input:** `BadLine1` `BadLine2`

`BadLine3` `The Matrix,TM001` `Action,SciFi,Thriller` - **Expected:** Skips 3 lines, parses one movie - **Variables Covered:** `i` (multiple redefinitions → P-uses in each iteration)

**TC-DF-MP-6:** `parseMovies_singleGenreToken_genresArrayLengthOne()` - **DU-Path:** genresArray.length > 1 → false → skip digit check - **Input:** `Matrix,M001` `Action` - **Expected:** Successfully parses with single genre - **Variables Covered:** `genresArray` (P-use length check false branch)

**TC-DF-MP-7:** `parseMovies_genresWithMultipleTokens_noDigit()` - **DU-Path:** genresArray.length > 1 → true, digit check → false - **Input:** `The Matrix,TM001` `Action,SciFi,Thriller` - **Expected:** Successfully parses with multiple genres - **Variables Covered:** `genresArray` (both P-uses: length true, digit false)

**TC-DF-MP-8:** `parseMovies_secondGenreTokenHasDigit_throws()` - **DU-Path:** genresArray[1].matches(".\d.") → true → throw - **Input:** `Movie,M001` `Action,Sc1Fi` - **Expected:** Throws exception - **Variables Covered:** `genresArray` (P-use in digit detection)

**TC-DF-MP-9:** `parseMovies_headerOnly_noGenresLine_throws()` - **DU-Path:** i++ after header → i >= lines.length → true - **Input:** `OnlyHeader,H001` - **Expected:** Throws "Genres are missing" - **Variables Covered:** `i` (definition → P-use at boundary check)

**TC-DF-MP-10:** `parseMovies_iIncrementedTwicePerIteration()` - **DU-Path:** i++ after header, i++ after genres → next iteration - **Input:** `Movie1,M001` `Action` `Movie2,M002` `Drama` - **Expected:** Parses 2 movies - **Variables Covered:** `i` (two redefinitions per iteration, reuse in next iteration)

---

#### 3.2.1.4 Coverage Summary for MovieParser

| Criterion | Coverage | Test Cases |
|---|---|---|
| All-Defs | 100% | All 10 tests |
| All-Uses | 100% | All 10 tests |
| All-DU-Paths | 100% | All 10 tests |

**Total Test Cases:** 10

---

### 3.2.2 3.2.2 UserParser.parseUsers() Data Flow Testing

**Method Signature:** `List<User> parseUsers(String usersFileData)`

**Complexity Factors:** - For-loop with non-standard step (`i += 2`) - Multiple exception branches based on input structure - Helper method call (`addLikedMovies`) with its own predicate + loop - Boundary check for liked movies line existence

**Reference:** `docs/data_flow/userparser-parseUsers.md`

---

#### 3.2.2.1 Variables Tracked

| Variable | Definition Points | Use Points |
|---|---|---|
| `lines` | `split("\n")` | Length access, array indexing |
| `length` | `lines.length` | Loop condition (P-use) |
| `i` | `= 0`, `i += 2` (multiple times) | Loop condition (P-use), array indexing (C-use), boundary checks (P-use) |
| `parts` | `lines[i].split(",")` | Length check (P-use), array access (C-use) |
| `user` | `new User(...)` | addLikedMovies parameter (C-use), list add (C-use) |

**Helper Method Variables (`addLikedMovies`):**

| Variable | Definition Points | Use Points |
|---|---|---|
| `parts` | `l.split(",")` | Digit check (P-use), iteration (C-use) |

---

**3.2.2.2 Definition-Use Paths** **DU-Path 1:** `i = 0` → P-use in `for (int i = 0; i < length; ...)`
**DU-Path 2:** `parts` definition → P-use in `if (parts.length != 2)`
**DU-Path 3:** `parts` definition → C-use in `new User(parts[0], parts[1], ...)`
**DU-Path 4:** `i` → P-use in `if (i+1 < length)`
**DU-Path 5:** `i` → C-use in `lines[i+1]` access
**DU-Path 6:** `i += 2` → P-use in next iteration's loop condition
**DU-Path 7:** Helper `parts` → P-use in digit regex check
**DU-Path 8:** Helper `parts` → C-use in for-each loop

---

**3.2.2.3 Test Cases** **Test Class:** `UserParserDataFlowTest`

**TC-DF-UP-1:** `parseUsers_pusePartsLengthNot2_true_throws()` - **DU-Path:** parts.length != 2 → true → throw exception - **Input:** `Alice,123456789,EXTRAM001` - **Expected:** Throws `UserException` "Invalid user data format" - **Variables Covered:** `parts` (P-use in length check), `i` (C-use in error message)

**TC-DF-UP-2:** `parseUsers_puseHasLikedLine_false_throws()` - **DU-Path:** (i+1 < length) → false → throw exception - **Input:** `Alice,123456789` - **Expected:** Throws "Liked movies are invalid for user 123456789" - **Variables Covered:** `i` and `length` (P-use in boundary check), `user` (C-use in message)

**TC-DF-UP-3:** `parseUsers_addLikedMovies_puseFirstTokenHasDigit_false_throws()` - **DU-Path:** parts[0].matches("$.\backslash d.$") → false → throw - **Input:** `Alice,123456789 MOVIE` - **Expected:** Throws exception (no digit in liked movies) - **Variables Covered:** Helper method `parts` (P-use in digit check)

**TC-DF-UP-4:** `parseUsers_emptyString_throws()` - **DU-Path:** Edge case - empty input - **Input:** `""` - **Expected:** Returns empty list or handles gracefully - **Variables Covered:** `lines` (definition → immediate use in length)

**TC-DF-UP-5:** `parseUsers_oneUser_singleIteration_coversIDefAndUse()` - **DU-Path:** i = 0 → one iteration → i += 2 → exit - **Input:** `Alice,123456789 M001,M002` - **Expected:** Returns 1 user with 2 liked movies - **Variables Covered:** `i` (initial definition → all uses in single iteration)

**TC-DF-UP-6:** `parseUsers_twoUsers_twoIterations_coversIReDefAndSecondIteration` - **DU-Path:** i = 0 → i = 2 → second iteration with different liked movies - **Input:** `Alice,123456789 M001,M002 Bob,987654321 M003,M004` - **Expected:** Returns 2 users with different liked movies - **Variables Covered:** `i` (redefinition via `i += 2` → reuse in second iteration)

**TC-DF-UP-7:** `parseUsers_pusePartsLengthEquals2_false_continues()` - **DU-Path:** parts.length == 2 → condition false → continue processing - **Input:** `Alice,123456789 M001` - **Expected:** Successful parsing - **Variables Covered:** `parts` (P-use with condition false)

**TC-DF-UP-8:** `parseUsers_partsLengthOne_throws()` - **DU-Path:** parts.length = 1 → != 2 → true → throw - **Input:** `AliceOnly M001` - **Expected:** Throws "Invalid user data format" - **Variables Covered:** `parts` (P-use with length 1)

**TC-DF-UP-9:** `parseUsers_partsLengthThree_throws()` - **DU-Path:** parts.length = 3 → != 2 → true → throw - **Input:** `Alice,123456789,Extra M001` - **Expected:** Throws "Invalid user data format" - **Variables Covered:** `parts` (P-use with length 3)

**TC-DF-UP-10:** `parseUsers_iPlusOneAtBoundary_validUser()` - **DU-Path:** (i+1 < length) exactly at boundary - **Input:** `Alice,123456789 M001` - **Expected:** Successful (i+1 exactly equals length at end) - **Variables Covered:** `i` (P-use at exact boundary)

**TC-DF-UP-11:** `parseUsers_addLikedMovies_singleId_oneIteration()` - **DU-Path:** Helper parts with one element → loop once - **Input:** `Alice,123456789 M001` - **Expected:** User with 1 liked movie - **Variables Covered:** Helper `parts` (C-use in single-iteration loop)

**TC-DF-UP-12:** `parseUsers_addLikedMovies_multipleIds_multipleIterations()` - **DU-Path:** Helper parts with multiple elements → loop multiple times - **Input:**

`Alice,123456789 M001,M002,M003,M004` - **Expected:** User with 4 liked movies - **Variables Covered:** Helper `parts` (C-use in multi-iteration loop)

**TC-DF-UP-13: `parseUsers_addLikedMovies_firstTokenNoDigit_throws()`** - **DU-Path:** parts[0] without digit → regex false → throw - **Input:** `Alice,123456789 INVALID` - **Expected:** Throws exception - **Variables Covered:** Helper `parts` (P-use with regex failing)

**TC-DF-UP-14: `parseUsers_addLikedMovies_firstTokenHasDigit_continues()`** - **DU-Path:** parts[0] with digit → regex true → continue - **Input:** `Alice,123456789 M001,M002` - **Expected:** Successful parsing - **Variables Covered:** Helper `parts` (P-use with regex passing)

---

#### 3.2.2.4 Coverage Summary for UserParser

| Criterion | Coverage | Test Cases |
|---|---|---|
| All-Defs | 100% | All 14 tests |
| All-Uses | 100% | All 14 tests |
| All-DU-Paths | 100% | All 14 tests |

**Total Test Cases:** 14

---

### 3.2.3 3.2.3 Recommender.recommendMovies() Data Flow Testing

**Method Signature:** `List<Movie> recommendMovies(User user)`

**Complexity Factors:** - Nested loops with multiple levels - De-duplication logic for genres (`contains()` predicate) - Early-exit logic (`break`) - Skip logic (`continue`) - Multiple derived data structures (`likedMovies`, `likedGenres`, `recommendations`)

**Reference:** `docs/data_flow/recommender-recommendMovies.md`

---

#### 3.2.3.1 Variables Tracked

| Variable | Definition Points | Use Points |
|---|---|---|
| `likedMoviesIDs` | `user.getLikedMovies()` | For-each loop (C-use) |
| `likedMovies` | `new ArrayList<>()` | add() calls (C-use), contains() (P-use), iteration (C-use) |

| Variable | Definition Points | Use Points |
|---|---|---|
| `likedGenres` | `new ArrayList<>()` | add() calls (C-use), contains() (P-use multiple times) |
| `genres` | `movie.getGenres()` | For-each iteration (C-use) |
| `genre` | Loop variable (2 contexts) | contains() checks (P-use), add() calls (C-use) |
| `recommendations` | `new ArrayList<>()` | add() calls (C-use), return (C-use) |
| `movie` | Loop variable (2 contexts) | Method calls (C-use), contains() (P-use) |

---

**3.2.3.2 Definition-Use Paths** **DU-Path 1:** `likedMoviesIDs` definition → C-use in first for-each loop
**DU-Path 2:** `likedMovies` definition → C-use in add() during first loop
**DU-Path 3:** `likedMovies` definition → C-use in iteration during second loop
**DU-Path 4:** `likedMovies` definition → P-use in contains() during third loop
**DU-Path 5:** `likedGenres` definition → P-use in `!likedGenres.contains(genre)` during construction
**DU-Path 6:** `likedGenres` definition → C-use in add() when genre is unique
**DU-Path 7:** `likedGenres` definition → P-use in `likedGenres.contains(genre)` during recommendation
**DU-Path 8:** `recommendations` definition → C-use in add() when match found
**DU-Path 9:** `recommendations` definition → C-use in return statement
**DU-Path 10:** `genre` definition (inner loop) → P-use in contains() checks

---

**3.2.3.3 Test Cases** **Test Class:** `RecommenderDataFlowTest`

**TC-DF-R-1:** `recommendMovies_emptyLikedMoviesIds_zeroIterationsInFirstLoops_ret` - **DU-Path:** likedMoviesIDs empty → 0 iterations → recommendations empty - **Setup:** - User with empty liked movies list - DataStore with available movies - **Expected:** Returns empty list - **Variables Covered:** `likedMoviesIDs` (definition → 0-iteration use), `recommendations` (definition → return with no adds)

**TC-DF-R-2:** `recommendMovies_exercises_skipLiked_continue_and_recommend_break()` - **DU-Path:** - likedMovies.contains(movie) → true → continue - likedGenres.contains(genre) → true → add + break - **Setup:** - User likes "The Matrix" (Action, Sci-Fi, Thriller) - DataStore: "The Matrix", "John Wick" (Action), "Finding Nemo" (Animation) - **Expected:** Recommends "John Wick", skips "The Matrix",

skips “Finding Nemo” - **Variables Covered:** - `likedMovies` (P-use in contains() → true branch) - `likedGenres` (P-use in contains() → true branch with break) - `recommendations` (C-use in add())

**TC-DF-R-3:** `recommendMovies_buildsLikedGenres_uniqueAddPath()` - **DU-Path:** !likedGenres.contains(genre) → true → add genre - **Setup:** - User likes movie with unique genres: “Matrix” (Action, Sci-Fi, Thriller) - **Expected:** likedGenres contains all 3 genres - **Variables Covered:** `likedGenres` (P-use in contains() → false, then C-use in add())

**TC-DF-R-4:** `recommendMovies_singleLikedMovie_oneIterationAllLoops()` - **DU-Path:** Size 1 lists → single iteration through all loops - **Setup:** - User likes “Matrix” (Action) - DataStore: “Matrix”, “John Wick” (Action) - **Expected:** Recommends “John Wick” - **Variables Covered:** All loops with exactly 1 iteration each

**TC-DF-R-5:** `recommendMovies_multipleLikedMovies_multipleIterations_multipleRe` - **DU-Path:** Multiple iterations building genres from multiple movies - **Setup:** - User likes “Matrix” (Action, Sci-Fi) and “Inception” (Action, Mystery) - DataStore includes candidates matching different genre subsets - **Expected:** Multiple recommendations based on merged genre set - **Variables Covered:** - `likedMovies` (multiple C-uses across iterations) - `likedGenres` (built from multiple movies) - `recommendations` (multiple adds)

**TC-DF-R-6:** `recommendMovies_duplicateGenres_skipsDuplicateAdd()` - **DU-Path:** !likedGenres.contains(genre) → false → skip add - **Setup:** - User likes movies with overlapping genres - “Matrix” (Action, Sci-Fi) and “John Wick” (Action, Thriller) - **Expected:** “Action” appears once in likedGenres - **Variables Covered:** `likedGenres` (P-use in contains() → true branch, no add)

**TC-DF-R-7:** `recommendMovies_noGenreMatches()` - **DU-Path:** Inner genre loop completes with no matches → no add - **Setup:** - User likes “Matrix” (Action) - DataStore only has “Finding Nemo” (Animation), “La La Land” (Musical) - **Expected:** Empty recommendations - **Variables Covered:** - `likedGenres` (P-use in inner contains() → always false) - `recommendations` (no adds, return empty)

**TC-DF-R-8:** `recommendMovies_multipleGenres_firstMatches()` - **DU-Path:** First genre in candidate matches → add + break (short-circuit) - **Setup:** - User likes “Matrix” (Action, Sci-Fi) - Candidate “John Wick” (Action, Thriller, Crime) - first genre matches - **Expected:** “John Wick” recommended, inner loop breaks early - **Variables Covered:** - `genre` (first iteration → P-use true → break) - `recommendations` (add on first match)

**TC-DF-R-9:** `recommendMovies_multipleGenres_laterMatches()` - **DU-Path:** First genres don’t match → later genre matches → add + break - **Setup:** - User likes “Matrix” (Sci-Fi) - Candidate has genres [Drama, Thriller, Sci-Fi] - third genre matches - **Expected:** Candidate recommended after checking multiple genres - **Variables Covered:** `genre` (multiple P-uses, eventually true)

**TC-DF-R-10:** `recommendMovies_allDUPathsCombined()` - **DU-Path:** Comprehensive test combining all paths - **Setup:** - Multiple liked movies with overlapping/unique genres - Multiple candidates: some liked, some matching, some not - **Expected:** Correct filtering and recommendation - **Variables Covered:** All tracked vari-

ables through all DU-paths

---

#### 3.2.3.4 Coverage Summary for Recommender

| Criterion | Coverage | Test Cases |
|---|---|---|
| All-Defs | 100% | All 10 tests |
| All-Uses | 100% | All 10 tests |
| All-DU-Paths | 100% | All 10 tests |

**Total Test Cases:** 10

---

### 3.2.4 3.2.4 Additional Data Flow Tests

**Test Class:** `UserDataFlowTest`

**Purpose:** Cover definition-use paths in the `User` class constructor, which contains complex validation logic.

**Variables Tracked:** - `name` (definition → P-uses in regex validations) - `id` (definition → P-uses in format and length validations) - Field assignments (C-uses)

**Total Test Cases:** 18

**Key Test Scenarios:** - Name validation: regex failures (starts with space, contains digits, mixed case) - ID format validation: regex failures (non-digit prefix, invalid characters) - ID length validation: boundary cases (8 chars, 10 chars) - Successful construction paths with various valid inputs

---

### 3.2.5 3.2.5 Data Flow Testing Summary

| Test Class | Target Method | Variables Tracked | Test Cases | Coverage |
|---|---|---|---|---|
| MovieParserDataFlowTest | parseMovies() | 9 variables | 10 | 100% All-DU-Paths |
| UserParserDataFlowTest | parseUsers() | 6 variables | 14 | 100% All-DU-Paths |
| RecommenderDataFlowTest | addMovie() | 7 variables | 10 | 100% All-DU-Paths |
| UserDataFlowTest | User constructor | 3 variables | 18 | 100% All-DU-Paths |

**Total Data Flow Tests:** 52 test methods

**Key Achievements:** - All variable definitions reach uses (All-Defs) - All uses of all definitions tested (All-Uses) - All definition-clear paths covered (All-DU-Paths) - Complex control flow thoroughly validated - Edge cases and boundary conditions tested

---

# 4 4. Overall Testing Summary

## 4.1 4.1 Test Distribution

| Testing Type | Test Classes | Test Methods | Coverage |
|---|---|---|---|
| Integration Testing | 3 | 5 | Component interactions |
| Path Coverage Testing | 1 | 8 | 100% statement/branch/condition |
| Data Flow Testing | 4 | 52 | 100% All-DU-Paths |
| **Total** | **8** | **65** | **Comprehensive** |

## 4.2 4.2 Testing Techniques Applied

### 4.2.1 Integration Testing (Top-Down)

- Tests component interactions
- Uses strategic mocking (Mockito)
- Validates workflow orchestration
- Verifies referential integrity across components

### 4.2.2 White-Box Testing: Coverage-Based

- Achieves 100% statement coverage
- Achieves 100% branch coverage
- Achieves 100% condition coverage
- Tests all control flow paths

### 4.2.3 White-Box Testing: Data Flow

- Achieves 100% All-Defs coverage
- Achieves 100% All-Uses coverage
- Achieves 100% All-DU-Paths coverage
- Tracks variables through complex control flow
- Tests variable redefinitions and reuses

## 4.3 4.3 Key Strengths

1. **Comprehensive Coverage:** Multiple complementary testing strategies ensure thorough validation

2. **Systematic Approach:** Each technique applied methodically with clear criteria
3. **Documentation:** Detailed documentation of paths, DU-paths, and test rationale
4. **Real-World Scenarios:** Tests include edge cases, boundary conditions, and error paths
5. **Maintainability:** Clear test names and structure facilitate maintenance

### 4.4 4.4 Testing Tools

- **JUnit 5.9.1:** Test framework
- **Mockito 5.8.0:** Mocking framework for integration tests
- **Maven Surefire:** Test execution
- **JaCoCo:** Coverage reporting

---

## 5 5. Conclusion

This project demonstrates a **professional-level software testing approach** that combines multiple testing strategies to ensure comprehensive quality assurance:

- **Integration testing** validates component interactions and workflow orchestration
- **Coverage-based testing** ensures all code paths are executed
- **Data flow testing** validates correct variable usage through all control flow paths

The combination of these techniques provides confidence in both the **functional correctness** and **structural completeness** of the Movie Recommendation System. The systematic application of testing criteria (statement/branch/condition coverage, All-Defs/Uses/DU-Paths) demonstrates rigorous software engineering practices.

**Total Test Cases:** 65 across 8 test classes
**Overall Coverage:** 100% across all metrics measured