White Box Testing Report

[CSE337s] Software Testing

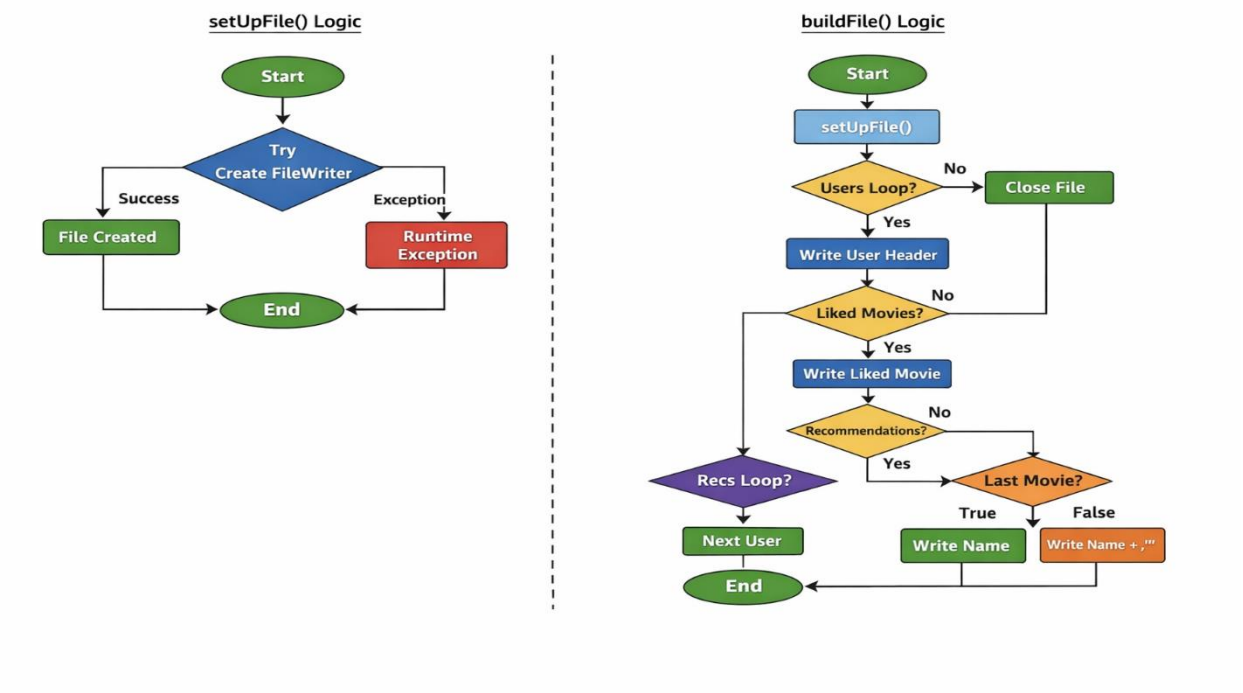
Team 7

Youssef Medhat Mahmoud	2200626
Mostafa Al Hassan Salah	2200747
Mario Milad Helmy	2200540
Ahmed Sayed Abdullah	2200737
Ibrahim Mahmoud Ibrahim	2200182
Omar Ahmed Mohamed	2200267
Seif Eldin Mustafa Abdel Fattah	2200794
Ahmed Saeed Abd Elhamid	2200689

Contents

WriteFile Class

1.Control Flow Graph (CFG)



2. Statement Coverage Analysis

Statement	Code Statement	Test Case	Executed
S1	WriteFile() default constructor	Object creation in all tests	Y
S2	Parameterized constructor	All tests	Y
S3	setUpFile() method call	All tests	Y
S4	new BufferedWriter(new FileWriter(filePath))	path_SetUpFile_Success	Y
S5	for(User user : users)	All buildFile tests	Y
S6	result.write(user.getName())	All buildFile tests	Y
S7	ArrayList<Movie> movieRec = user.getRecommendedMovies()	All buildFile tests	Y

Statement	Code Statement	Test Case	Executed
S8	for(Movie movie : moviesLiked)	All buildFile tests	Y
S9	if(movieRec.size() != 0)	Empty & non-empty recommendation tests	Y
S10	for(Movie movie : movieRec)	path_WithRecommendations tests	Y
S11	if(counter == moviesLiked.size()-1)	Multiple iteration tests	Y
S12	result.close()	All buildFile tests	Y

Statement Coverage Result: 12 / 12 statements executed, 100% Statement Coverage

3. Path Coverage Analysis

Path ID	Path Description	Test Method
Path 1	setUpFile() → SUCCESS (file created successfully)	path_SetUpFile_Success
Path 2	setUpFile() → EXCEPTION (invalid path)	path_SetUpFile_Exception
Path 3	users ≠ empty, movieRec.size() == 0 (no recommendations)	path_NoRecommendations
Path 4	movieRec.size() ≠ 0, counter == last index (TRUE) - single recommendation	path_WithRecommendations_LastIndexOnly
Path 5	movieRec.size() ≠ 0, counter FALSE → TRUE (multiple iterations)	path_WithRecommendations_MultipleIterations

4. Branch Coverage Analysis

Branch	Decision	True Branch	False Branch	Test Case(s)	Covered
B1	File creation success	File created	Exception thrown	path_SetUpFile_Success, path_SetUpFile_Exception	Y

Branch	Decision	True Branch	False Branch	Test Case(s)	Covered
B2	users loop	Users exist	No users	All tests	Y
B3	movieRec.size() != 0	Has recommendations	No recommendations	path_NoRecommendations, path_WithRecommendations tests	Y
B4	counter == last	Last movie	Not last movie	path_WithRecommendations_MultipleIterations	Y
B5	Recommendation loop	Loop entered	Loop skipped	path_NoRecommendations, path_WithRecommendations tests	Y

Branch Coverage Result: All branches evaluated true and false, 100% Branch Coverage

5. WriteFile Coverage Summary

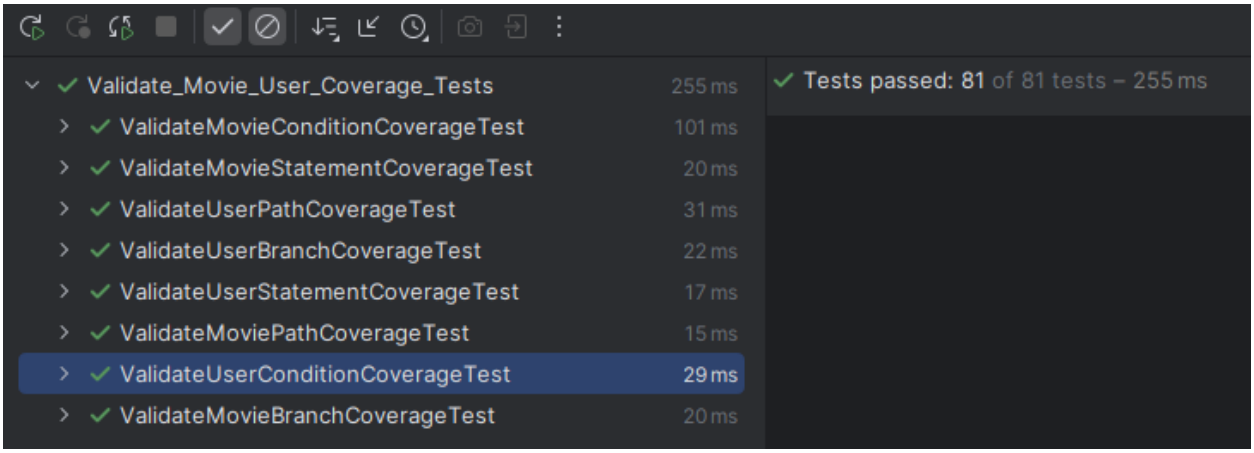
Coverage Type	Status
Statement Coverage	**100%**
Branch Coverage	**100%**
Path Coverage	**Complete**

ValidateUser and ValidateMovie Classes

This report summarizes the white-box testing performed on the `ValidateUser` and `ValidateMovie` classes. The objective was to ensure structural integrity by exercising every internal code path, decision point, and individual statement.

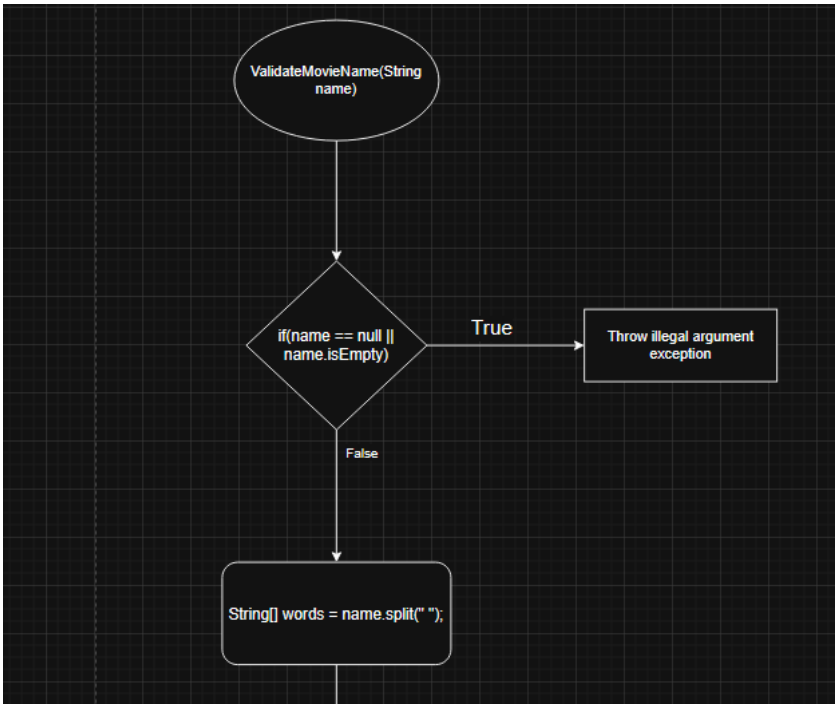
The testing strategy employed four primary coverage criteria:

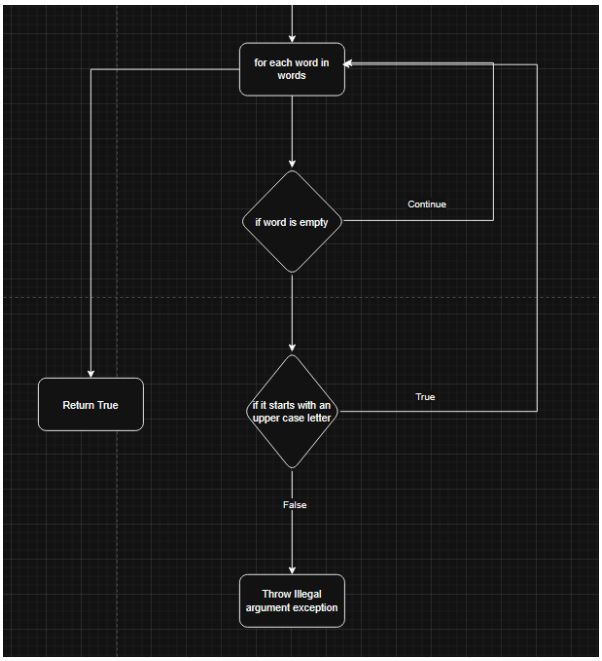
- **Statement Coverage:** Executing all executable lines.
- **Branch Coverage:** Traversing all decision outcomes (True/False).
- **Condition Coverage:** Evaluating all sub-conditions in compound decisions.
- **Path Coverage:** Executing all possible end-to-end flows.



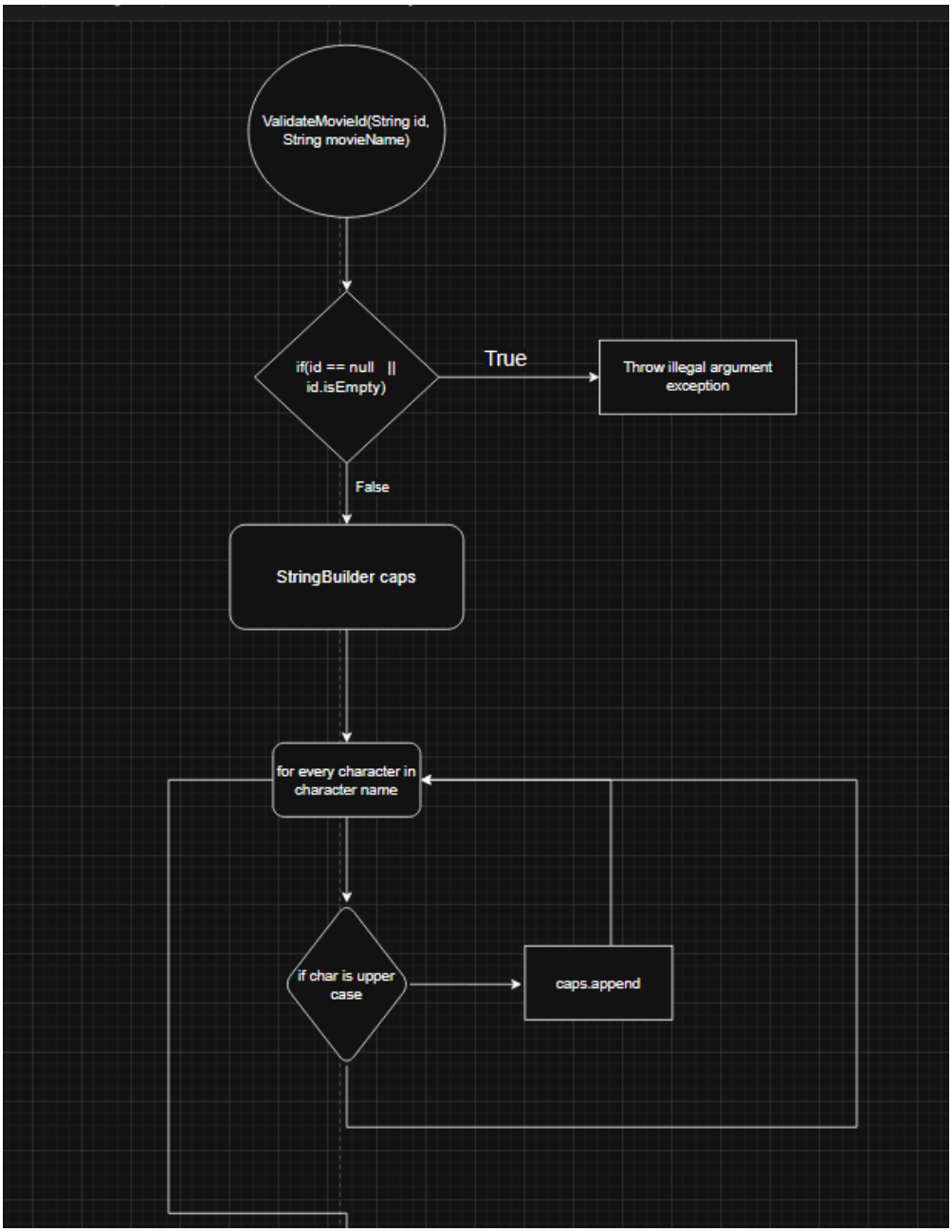
Control Flow Graphs

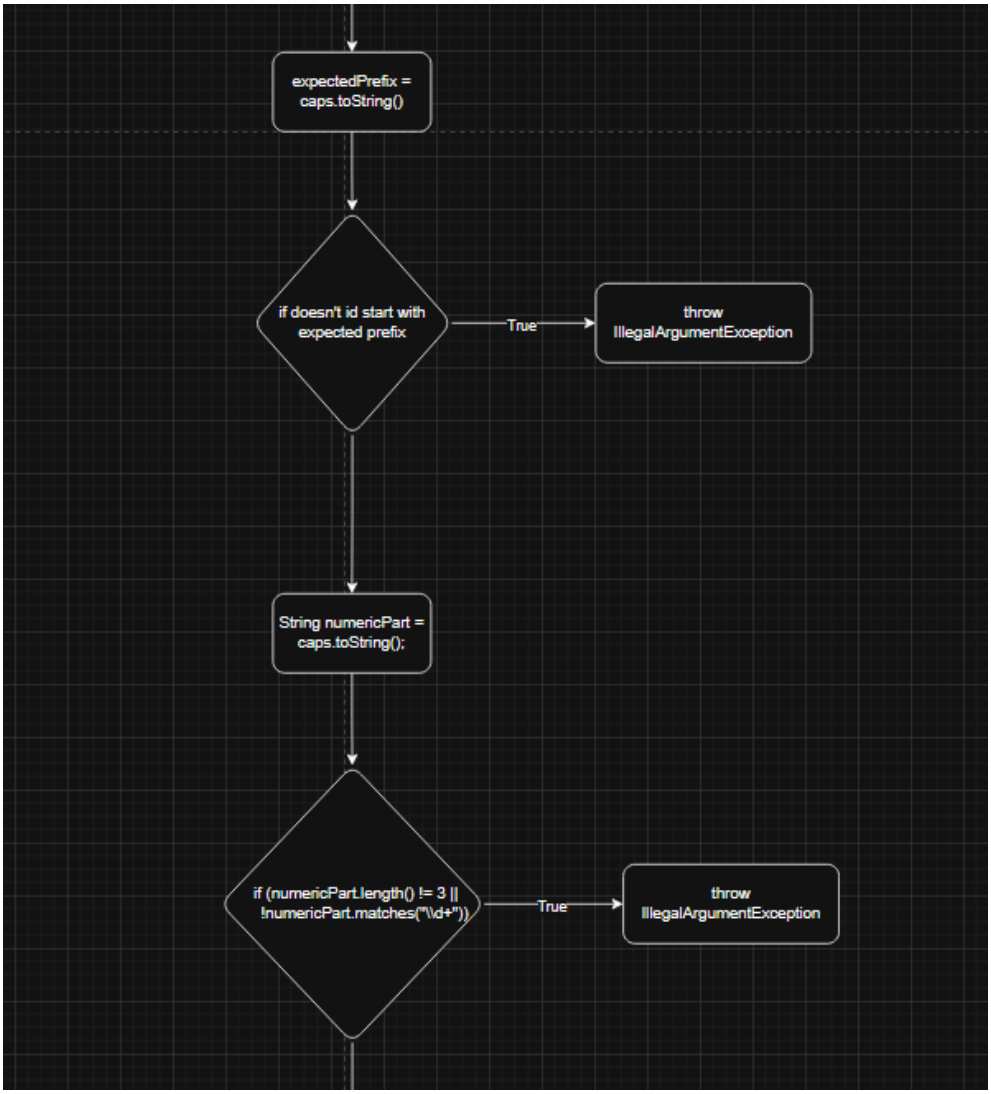
ValidateMovieName

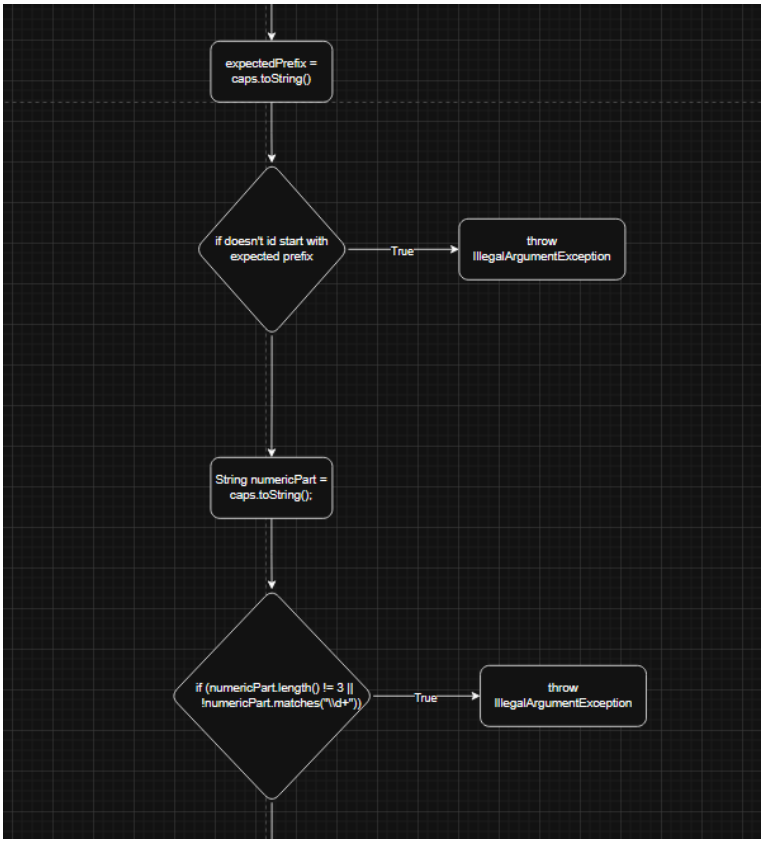


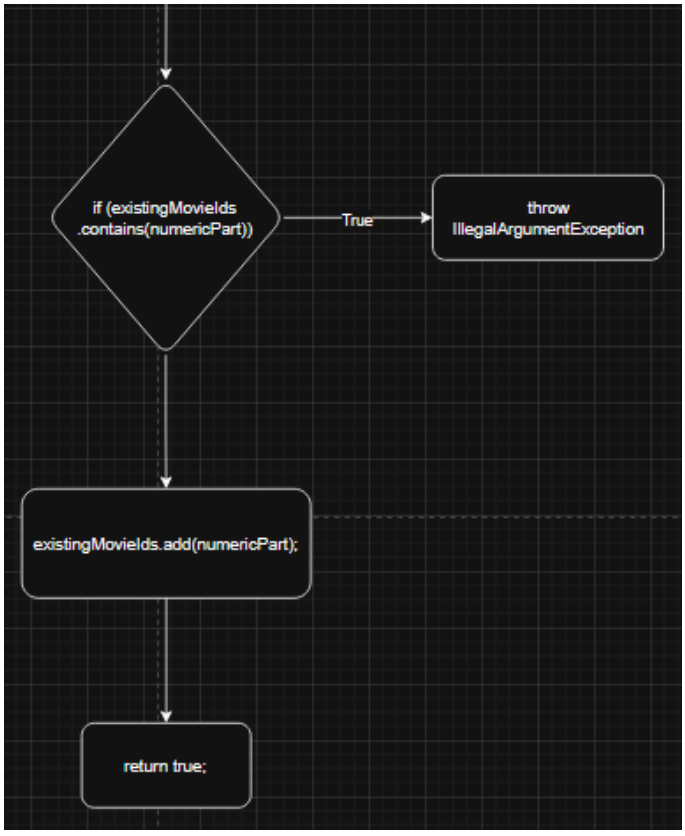


ValidateMovieId

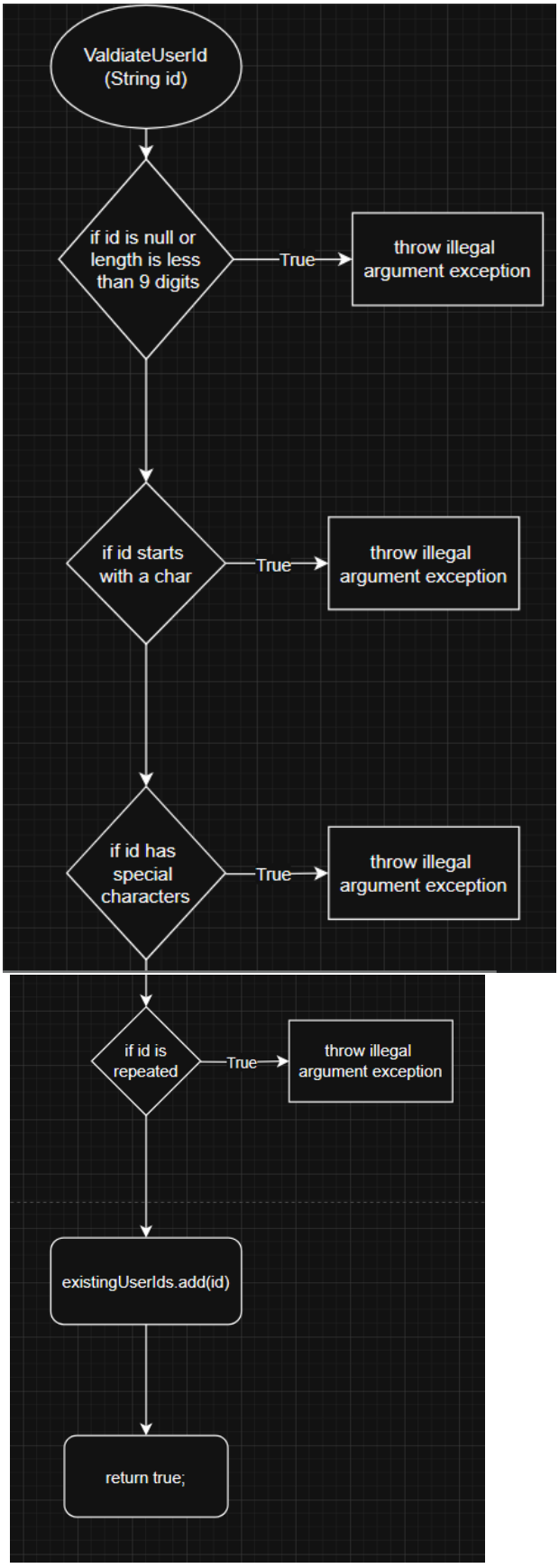



expectedPrefix = caps.toString()
if doesn't id start with expected prefix
True
throw IllegalArgumentException
String numericPart = caps.toString();
if (numericPart.length() != 3 || !numericPart.matches("\\d+"))
True
throw IllegalArgumentException

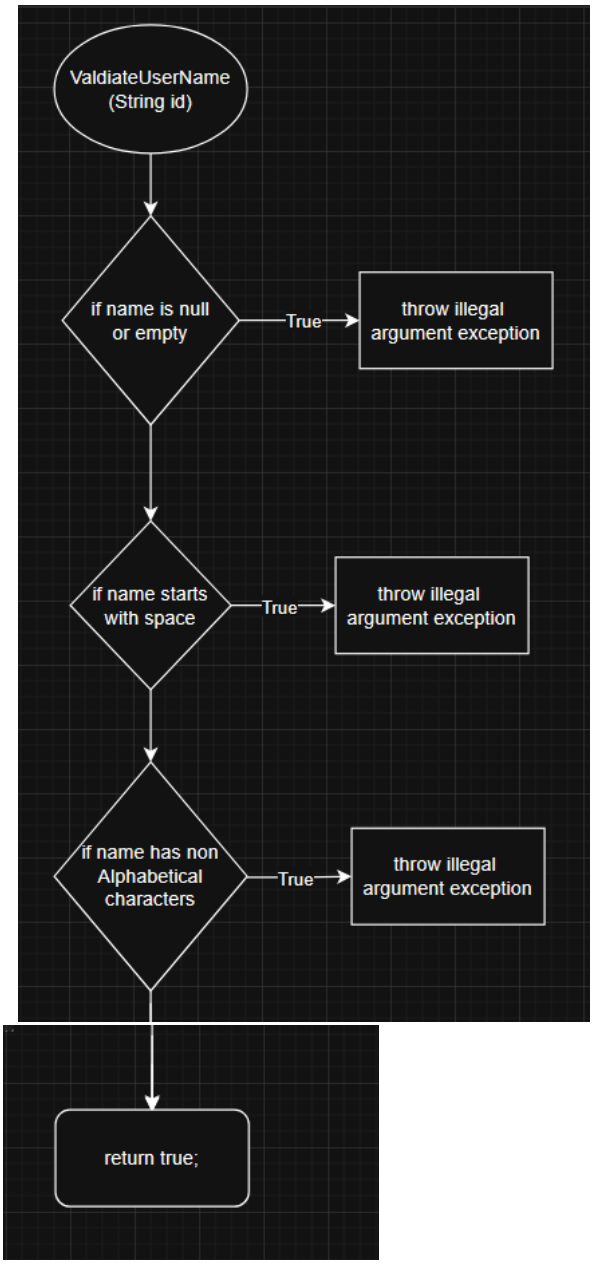

expectedPrefix = caps.toString()
if doesn't id start with expected prefix
True
throw IllegalArgumentException
String numericPart = caps.toString();
if (numericPart.length() != 3 || !numericPart.matches("\\d+"))
True
throw IllegalArgumentException


if (existingMovieIds .contains(numericPart))
True
throw IllegalArgumentException
existingMovieIds.add(numericPart);
return true;

ValidateUserId



ValidateUserName



ValidateUser Class Coverage Summary

The `ValidateUser` class uses a set of existing user IDs for uniqueness checks. Inferred method logic:
- `validateUserId`: Checks for non-null, length=9, starts with digit, alphanumeric (no special chars), and unique.
- `validateUserName`: Checks for non-null/non-empty, doesn't start with space, contains only letters and spaces.

Coverage Metrics Overview

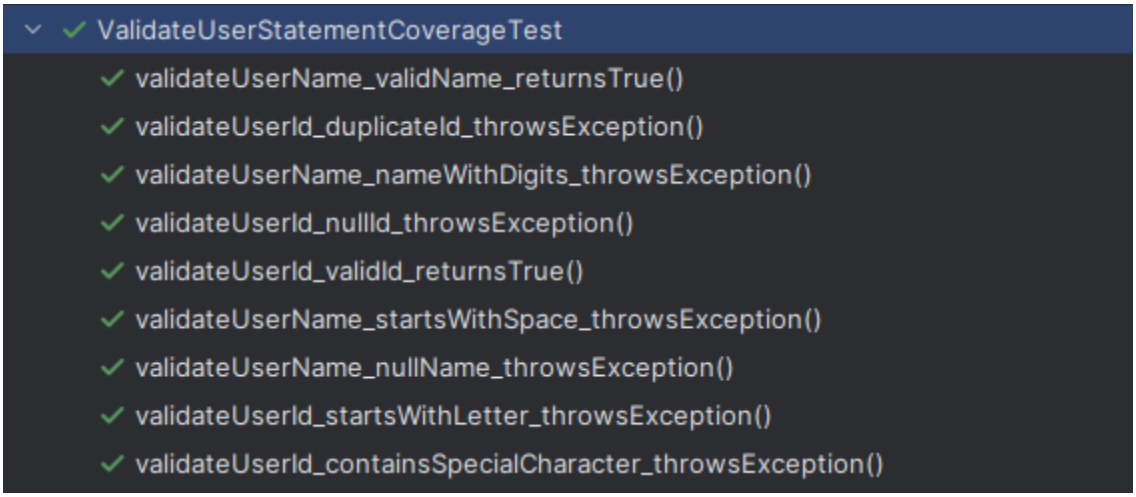
Coverage Type	**validateUserId Achievement**	**validateUserName Achievement**	**Test Class File**
Statement	10/10(100%)	7/7(100%)	ValidateUserStatementCoverageTest.java
Branch	10/10(100%)	10/10(100%)	ValidateUserBranchCoverageTest.java
Condition	All conditions covered (inferred 100%)	All conditions covered (inferred 100%)	ValidateUserConditionCoverageTest.java
Path	5/5paths (100%)	5/5paths (100%)	ValidateUserPathCoverageTest.java

Statement Coverage (ValidateUserStatementCoverageTest.java)

This suite incrementally covers statements by adding tests that execute new lines of code. Statements likely include checks for null, length, first char, pattern match, and uniqueness.

- **validateUserId**:
  - Initial test (`validateUserId_validId_returnsTrue`): Covers 6/10 statements (valid path).
  - `validateUserId_nullId_throwsException`: Covers +1 (null check), total 7/10.
  - `validateUserId_startsWithLetter_throwsException`: Covers +1 (first char check), total 8/10.
  - `validateUserId_containsSpecialCharacter_throwsException`: Covers +1 (pattern check), total 9/10.
  - `validateUserId_duplicateId_throwsException`: Covers +1 (uniqueness check), total 10/10.
- **validateUserName**:
  - Initial test (`validateUserName_validName_returnsTrue`): Covers 4/7 statements (valid path).
  - `validateUserName_nullName_throwsException`: Covers +1 (null check), total 5/7.
  - `validateUserName_startsWithSpace_throwsException`: Covers +1 (space check), total 6/7.
  - `validateUserName_nameWithDigits_throwsException`: Covers +1 (pattern check), total 7/7.

Achieves full statement coverage with minimal tests. Without redundant tests.



Branch Coverage (ValidateUserBranchCoverageTest.java)

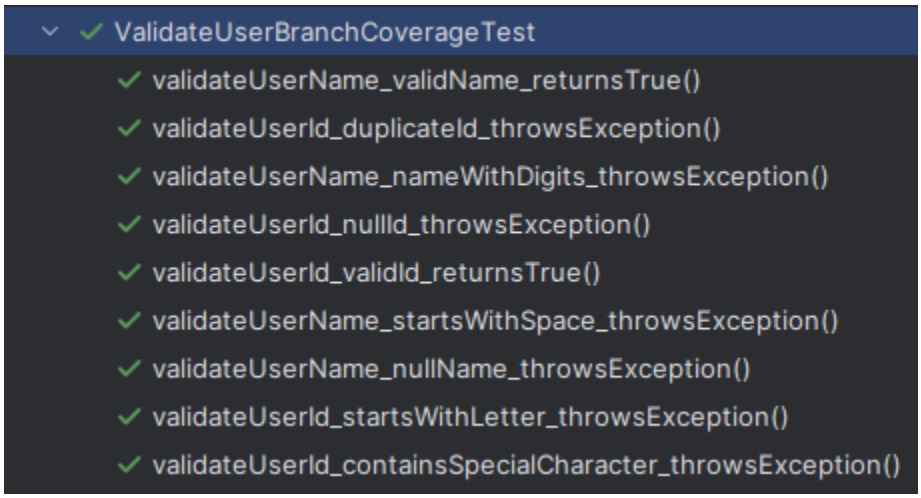
This suite targets decision branches (true/false outcomes of if-statements).

validateUserId:

- Initial test (`validateUserId_validId_returnsTrue`): Covers 6/10 branches.
- `validateUserId_nullId_throwsException`: Covers +1, total 7/10.
- `validateUserId_startsWithLetter_throwsException`: Covers +1, total 8/10.
- `validateUserId_containsSpecialCharacter_throwsException`: Covers +1, total 9/10.
- `validateUserId_duplicateId_throwsException`: Covers +1, total 10/10.

validateUserName:

- Initial test (`validateUserName_validName_returnsTrue`): Covers 4/7 branches.
- `validateUserName_nullName_throwsException`: Covers +1, total 5/7.
- `validateUserName_startsWithSpace_throwsException`: Covers +1, total 6/7.
- `validateUserName_nameWithDigits_throwsException`: Covers +1, total 7/7.



Condition Coverage (ValidateUserConditionCoverageTest.java)

This suite evaluates all sub-conditions in compound decisions (e.g., A || B, where A and B are tested true/false independently). Conditions are explicitly listed in the file.

ValidateUserId method

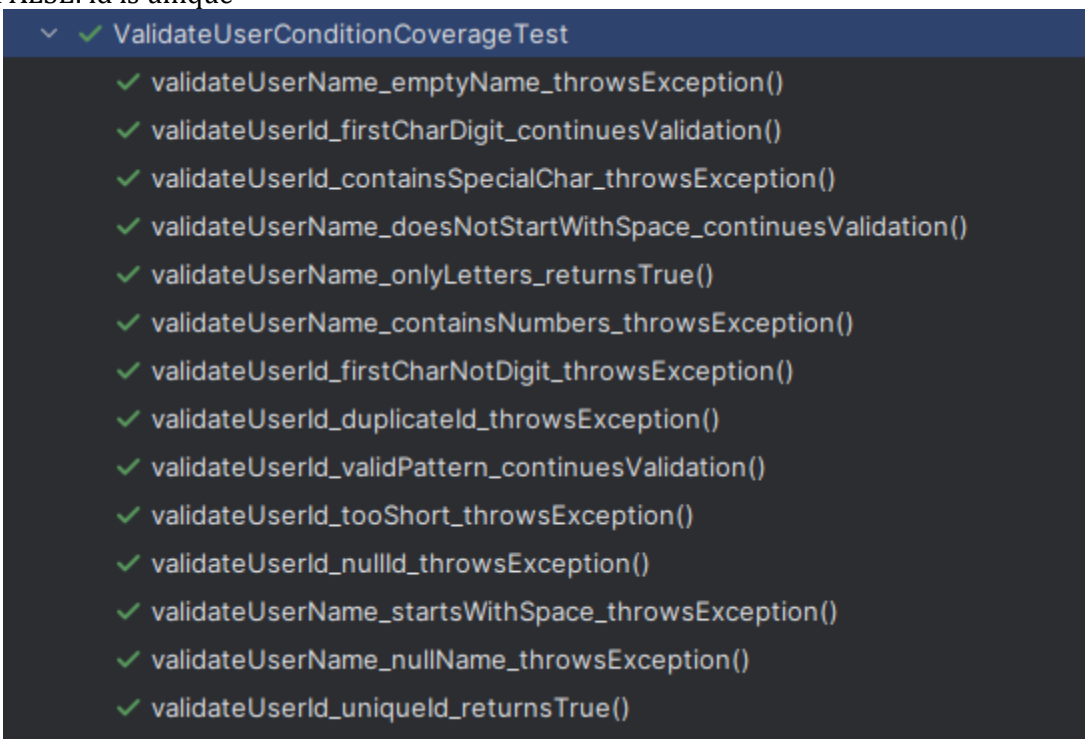
Condition A: id == null || id.length() != 9

A1: id == null (TRUE/FALSE)
A2: id.length() != 9 (TRUE/FALSE)
Condition B: !Character.isDigit(id.charAt(0))

TRUE: first character is NOT a digit
FALSE: first character IS a digit
Condition C: !id.matches("[0-9][a-zA-Z0-9]*")

TRUE: id doesn't match pattern (has special chars)
FALSE: id matches pattern
Condition D: existingUserIds.contains(id)

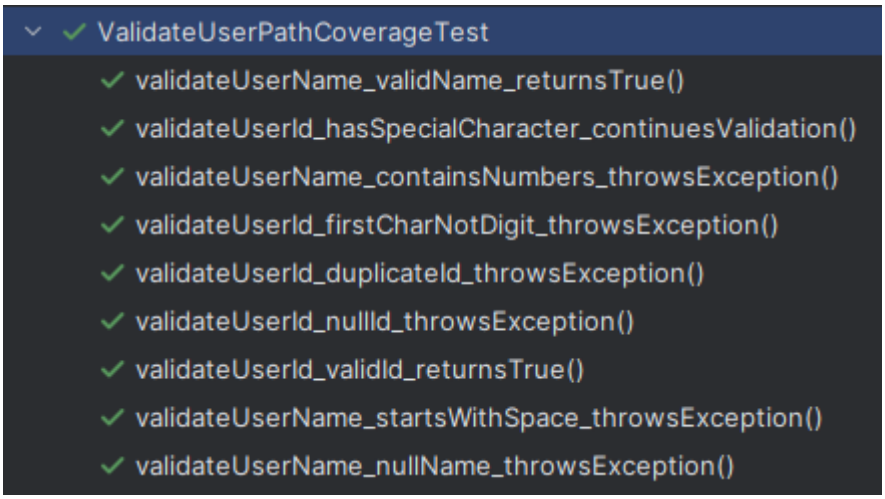
TRUE: id already exists
FALSE: id is unique



Path Coverage (ValidateUserPathCoverageTest.java)

This suite targets distinct execution paths (combinations of branches).

- **validateUserId**: 5 paths covered.
  - Path 1: Null input.
  - Path 2: Invalid first char.
  - Path 3: Special char.
  - Path 4: Duplicate.
  - Path 5: Valid.
- **validateUserName**: 4 paths covered.
  - Path 1: Null.
  - Path 2: Starts with space.
  - Path 3: Contains numbers.
  - Path 4: Valid.



ValidateMovie Class Coverage Summary

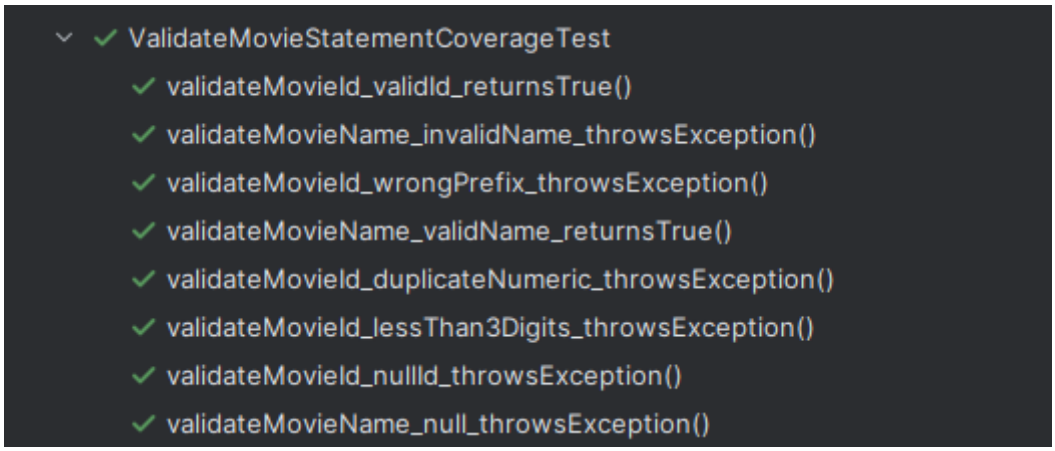
The `ValidateMovie` class uses a set of existing movie numeric IDs for uniqueness. Inferred method logic:
- `ValidateMovieName`: Checks non-null/non-empty, splits by space, skips empty words, ensures each word starts uppercase.
- `ValidateMovieId`: Checks non-null/non-empty, derives prefix from name initials (uppercase letters), ensures 3-digit numeric suffix, unique numeric part.

Coverage Metrics Overview

Coverage Type	ValidateMovieName Achievement	ValidateMovieId Achievement	Test Class File
Statement	8/8 (100%)	16/16 (100%)	ValidateMovieStatementCoverageTest.java
Branch	10/10(100%)	18/18 (100%)	ValidateMovieBranchCoverageTest.java
Condition	All conditions covered (100%)	All conditions covered (100%)	ValidateMovieConditionCoverageTest.java
Path	3/3 paths (100%)	5/5 paths (100%)	ValidateMoviePathCoverageTest.java

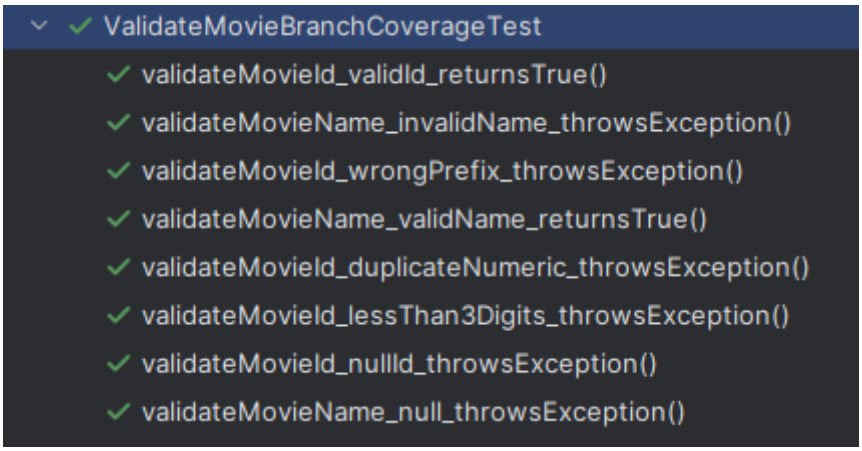
Statement Coverage (ValidateMovieStatementCoverageTest.java)

- **ValidateMovieName**:
  - Initial test (`validateMovieName_validName_returnsTrue`): Covers 6/8 statements.
  - `validateMovieName_null_throwsException`: Covers +1, total 7/8.
  - `validateMovieName_invalidName_throwsException`: Covers +1 (lowercase check), total 8/8.
- **ValidateMovieId**:
  - Initial test (`validateMovieId_validId_returnsTrue`): Covers 12/16 statements.
  - `validateMovieId_nullId_throwsException`: Covers +1, total 13/16.
  - `validateMovieId_wrongPrefix_throwsException`: Covers +1, total 14/16.
  - `validateMovieId_lessThan3Digits_throwsException`: Covers +1, total 15/16.
  - `validateMovieId_duplicateNumeric_throwsException`: Covers +1, total 16/16.



Branch Coverage (ValidateMovieBranchCoverageTest.java)

- **ValidateMovieName**:
  - Initial test (`validateMovieName_validName_returnsTrue`): Covers 8/10 branches.
  - `validateMovieName_null_throwsException`: Covers +1, total 9/10.
  - `validateMovieName_invalidName_throwsException`: Covers +1, total 10/10.
- **ValidateMovieId**:
  - Initial test (`validateMovieId_validId_returnsTrue`): Covers 14/18 branches.
  - `validateMovieId_nullId_throwsException`: Covers +1, total 15/18.
  - `validateMovieId_wrongPrefix_throwsException`: Covers +1, total 16/18.
  - `validateMovieId_lessThan3Digits_throwsException`: Covers +1, total 17/18.
  - `validateMovieId_duplicateNumeric_throwsException`: Covers +1, total 18/18.



Condition Coverage (ValidateMovieConditionCoverageTest.java)

ValidateMovieName

• Condition A: name == null || name.isEmpty()

- Subcondition A1: name == null (TRUE/FALSE)
- Subcondition A2: name.isEmpty() (TRUE/FALSE)

• Condition B: word.isEmpty() (in loop)

- TRUE: word is empty
- FALSE: word is not empty

• Condition C: !Character.isUpperCase(word.charAt(0))

- TRUE: first character is NOT uppercase
- FALSE: first character IS uppercase

ValidateMovieId

Condition A: id == null || id.isEmpty()

- A1: id == null (TRUE/FALSE)
- A2: id.isEmpty() (TRUE/FALSE)

Condition B: Character.isUpperCase(c)

- TRUE: character is uppercase
- FALSE: character is NOT uppercase

Condition C: !id.startsWith(expectedPrefix)

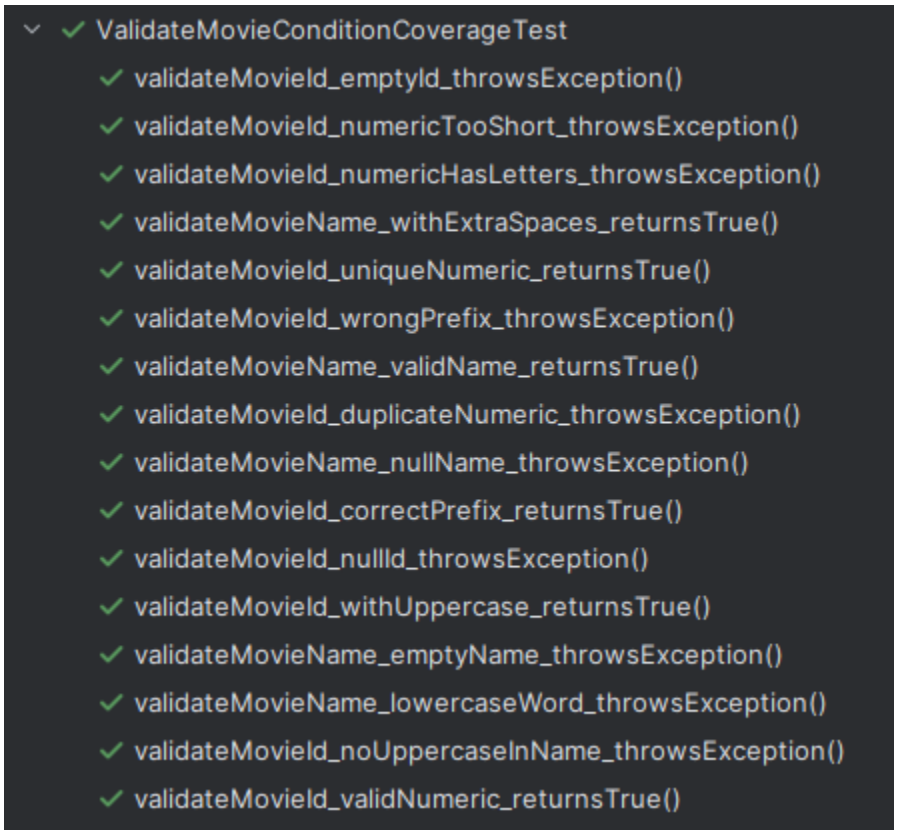
- TRUE: id doesn't start with prefix
- FALSE: id starts with prefix

Condition D: numericPart.length() != 3 || !numericPart.matches("\d+")

- D1: numericPart.length() != 3 (TRUE/FALSE)
- D2: !numericPart.matches("\d+") (TRUE/FALSE)

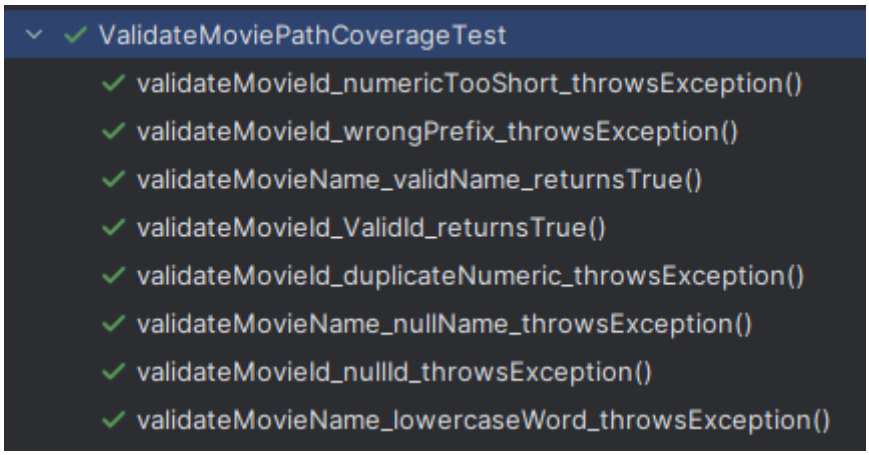
Condition E: existingMovieIds.contains(numericPart)

- TRUE: numeric part already exists
- FALSE: numeric part is unique


ValidateMovieConditionCoverageTest
validateMovieId_emptyId_throwsException()
validateMovieId_numericTooShort_throwsException()
validateMovieId_numericHasLetters_throwsException()
validateMovieName_withExtraSpaces_returnsTrue()
validateMovieId_uniqueNumeric_returnsTrue()
validateMovieId_wrongPrefix_throwsException()
validateMovieName_validName_returnsTrue()
validateMovieId_duplicateNumeric_throwsException()
validateMovieName_nullName_throwsException()
validateMovieId_correctPrefix_returnsTrue()
validateMovieId_nullId_throwsException()
validateMovieId_withUppercase_returnsTrue()
validateMovieName_emptyName_throwsException()
validateMovieName_lowercaseWord_throwsException()
validateMovieId_noUppercaseInName_throwsException()
validateMovieId_validNumeric_returnsTrue()

Path Coverage (ValidateMoviePathCoverageTest.java)

- **ValidateMovieName**: 3 paths covered.
  - Path 1: Valid.
  - Path 2: Null.
  - Path 3: Lowercase word.
- **ValidateMovieId**: 5 paths covered.
  - Path 1: Valid.
  - Path 2: Null.
  - Path 3: Wrong prefix.
  - Path 4: Numeric too short.
  - Path 5: Duplicate numeric.



Data Flow Testing

For the validator classes there is no need for data flow testing since the code contains only single definitions and no branching between definitions and uses. Any branching leads to exceptions which terminate execution rather than creating alternative data flow paths, and equivalent coverage is already achieved through statement, branch, and condition testing.

ReadUser class

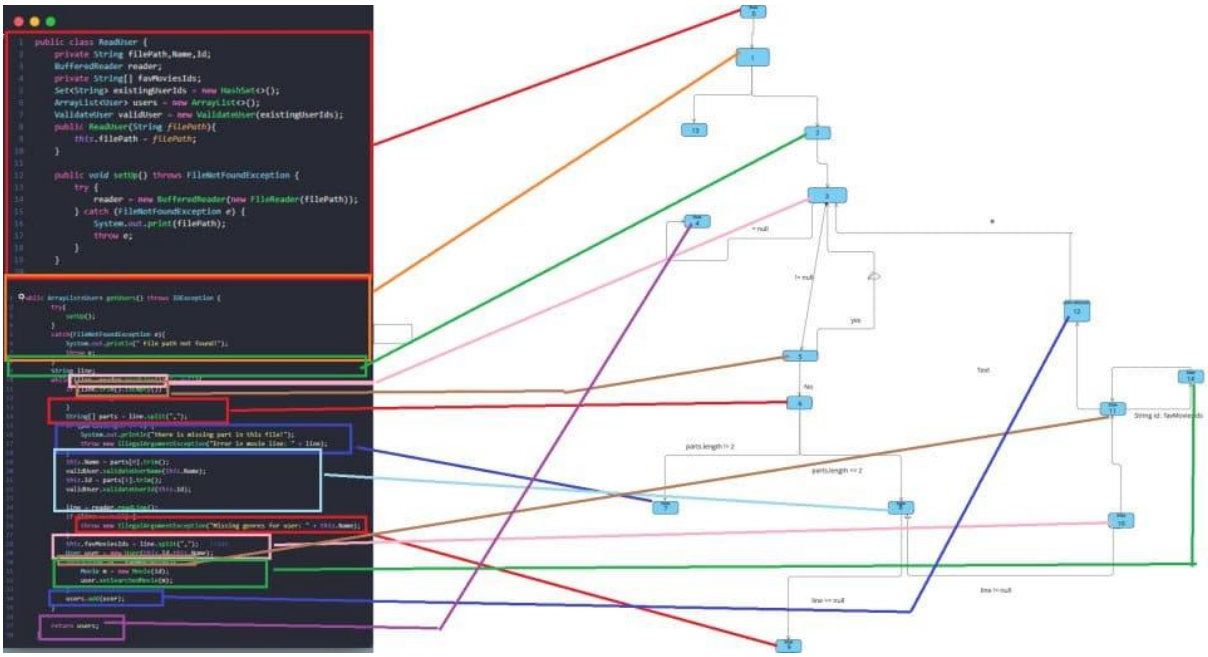
Data Flow Graph (DFG) by All-DU

Overview

The ReadUser class is designed to process user data from an external file. It handles file initialization, parses line-by-line information (including names, IDs, and movie genres) and performs validation. To ensure the robustness of this class, we have mapped the code to a **Data Flow Graph (DFG)**.

➢ The provided DFG maps the control flow and data dependencies within the getUsers() method. This graph is essential for **White-Box Testing**, allowing us to identify all possible execution paths.

- **Nodes & Mapping:** Each colored block in the source code corresponds to a numbered node in the graph.
    - **Node 0-2:** Initialization and file setup.
    - **Node 3:** The primary loop and null-check condition (line != null).
    - **Node 4:** Return Users.
    - **Nodes 5-8:** The core logic for splitting strings and validating specific fields (Name and ID).
    - **Nodes 8-14:** Handling of genres and adding the final User object to the list.
    - **Node 13:** Handling File Exception.
- The graph illustrates the branching logic where the program might throw an IllegalArgumentException if data is missing or malformed, ensuring that every "Edge" (transition) in the code is accounted for.

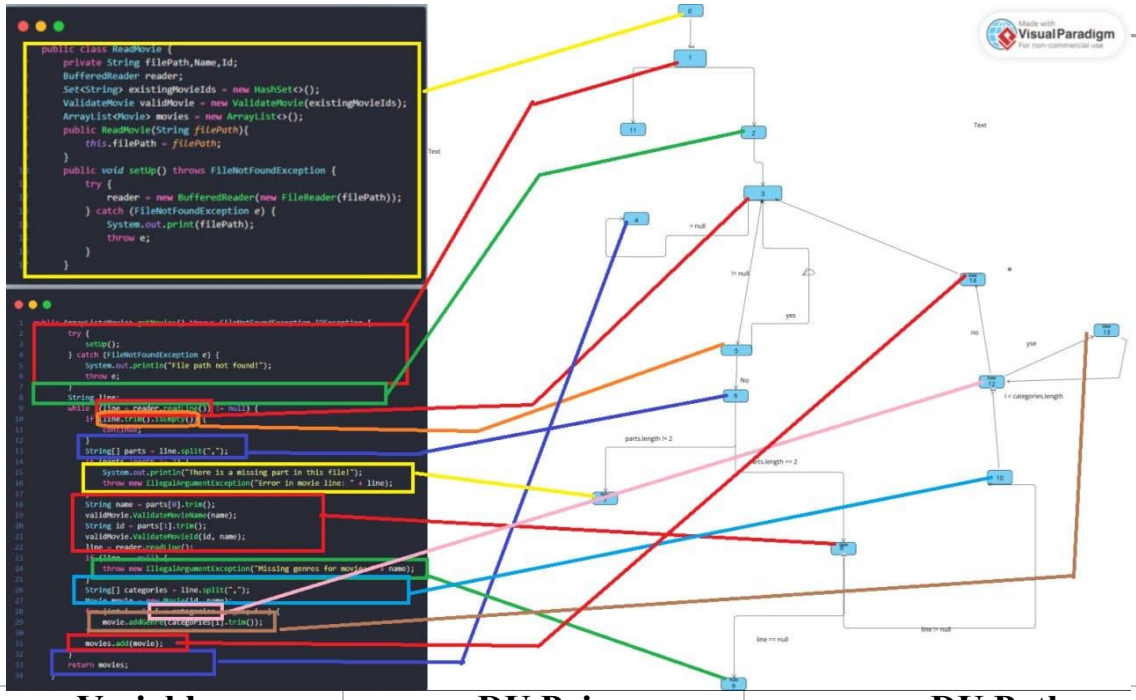


Variable	DU Pair	DU Path
validUser	(0,8)	(0,1,2,3,5,6,8)
reader	(0,3) (0,8)	(0,1,2,3), (0,1,2,3,5,6,8)
favMoviesIds	(10,11)	(10,11)
existingUserIds	(0,0)	No Path needed
users	(0,12),(0,4)	(0,1,1,2,3,5,6,8,10,11,14,14,12), (0,1,2,3,4)
Line	(3,5),(3,6),(8,10),(3,7) ,(3,(3,4)),(3,(3.5)) ,(8,(8,9)),(8,(8.10))	(3,5), (3,5,6), (8,10) (3,4), (8,9)
Parts	(6,8),(6,(6,8)),(6,(6,7))	(6,7), (6,8), (6,7)
name	(8,9),(8,10)	(8,9), (8,10), (8,9)
Id	(8,10)	(8,10)
User	(10,12),(10,14)	(10,11,14), (10,11,14,11,14,12)
id	(11,14)	(11,14)
m	(14,14)	No Path needed
filepath	(0,1)	(0,1)

1.1 Overview

The ReadUser class is designed to process user data from an external file. It handles file initialization, parses line-by-line information (including names, IDs, and movie genres) and performs validation. To ensure the robustness of this class, we have mapped the code to a **Data Flow Graph (DFG)**.

- The provided DFG maps the control flow and data dependencies within the getMovies() method. This graph is essential for **White-Box Testing**, allowing us to identify all possible execution paths.
- **Nodes & Mapping:** Each colored block in the source code corresponds to a numbered node in the graph.
    - **Node 0-2:** Initialization and file setup.
    - **Node 3:** The primary loop and null-check condition (line != null).
    - **Node 4:** Return Movies.
    - **Nodes 5-8:** The core logic for splitting strings and validating specific fields (Name and ID).
    - **Nodes 8-14:** Handling of genres and adding the final User object to the list.
    - **Node 13:** Handling File Exception.
- The graph illustrates the branching logic where the program might throw an IllegalArgumentException if data is missing or malformed, ensuring that every "Edge" (transition) in the code is accounted for.

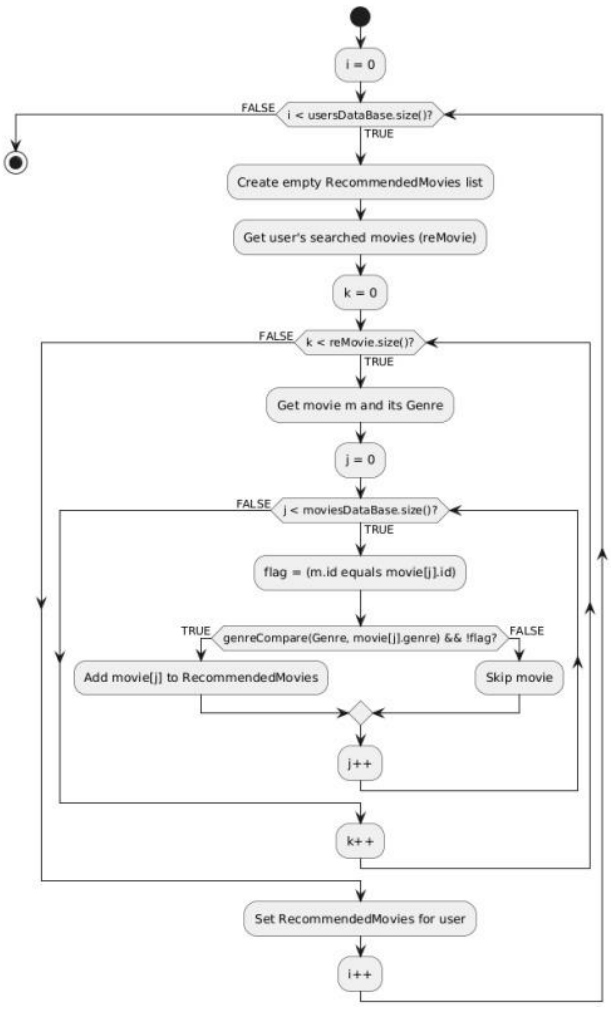


Variable	DU Pair	DU Path
Movies	(0,4), (0,14)	(0,1,1,2,3,5,6,8,10,12,13,12,14) (0,1,2,3,4)
Line	(3,5), (3,6), (8,10) ,(3,(3,4)), (3,(3.5)) ,(8,(8,9)),(8,(8.10))	(3,5), (3,5,6), (8,10) (3,4), (8,9)
Parts	(6,8),(6,(6,8)),(6,(6,7))	(6,7),(6,8)
name	(8,9),(8,10)	(8,9),(8,10)
id	(8,10)	(8,10)
Categories	(10,12),(10,13), (10,(12,13)),(10,(12,14)	(10,12),(10,12,13),(10,12,13,12,14)
Movie	(10,13),(10,14)	(10,12,13),(10,12,13,12,14) ,(10,12,13,12,14)
ValidMovie	(0,8)	(0,1,2,3,5,6,8)
reader	(0,3)(0,8)	(0,1,2,3), (0,1,2,3,5,6,8)
existingUserIds	(0,0)	No Path needed
filePath	(0,1)	(0,1)
Movies	(0,4),(0,14)	(0,1,1,2,3,5,6,8,10,12,13,12,14) (0,1,2,3,4)
Line	(3,5), (3,6), (8,10) , (3, (3, 4)), (3, (3.5)) , (8, (8, 9)), (8, (8.10))	(3,5), (3,5,6), (8,10) (3,4), (8,9)

Data Base Class

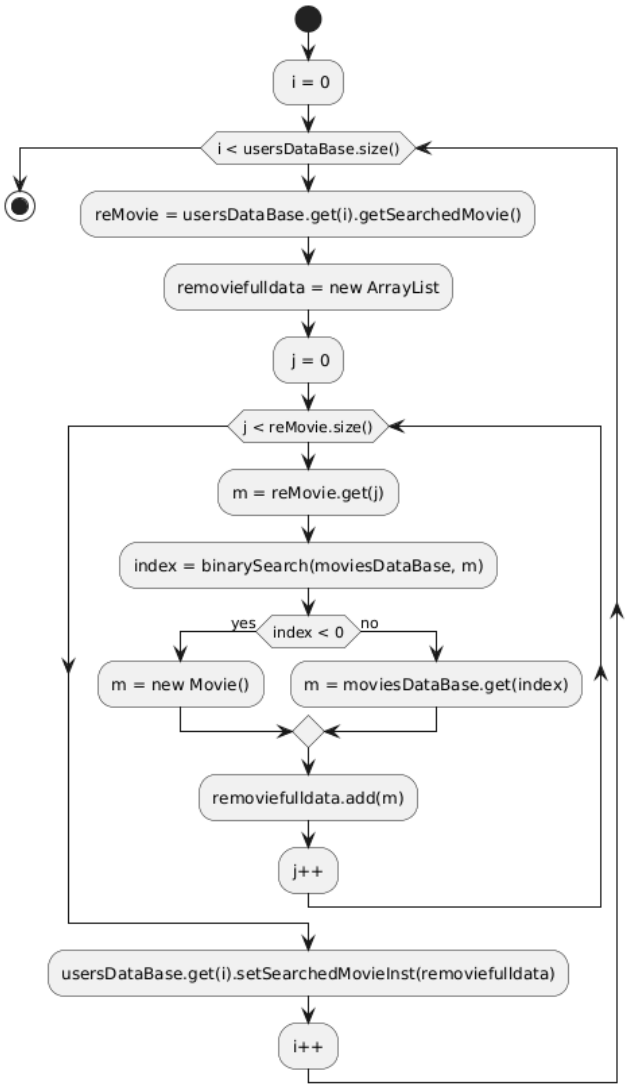
UML code for movie recommendation

```
@startuml movieRecommend_Activity
start
:i = 0;
while (i < usersDataBase.size()?) is (TRUE)
  :Create empty RecommendedMovies list;
  :Get user's searched movies (reMovie);
  :k = 0;
  while (k < reMovie.size()?) is (TRUE)
    :Get movie m and its Genre;
    :j = 0;
    while (j < moviesDataBase.size()?) is (TRUE)
      :flag = (m.id equals movie[j].id);
      if (genreCompare(Genre, movie[j].genre) && !flag?) then (TRUE)
        :Add movie[j] to RecommendedMovies;
      else (FALSE)
        :Skip movie;
      endif
      :j++;
    endwhile (FALSE)
    :k++;
  endwhile (FALSE)
  :Set RecommendedMovies for user;
  :i++;
endwhile (FALSE)
stop
@enduml
```



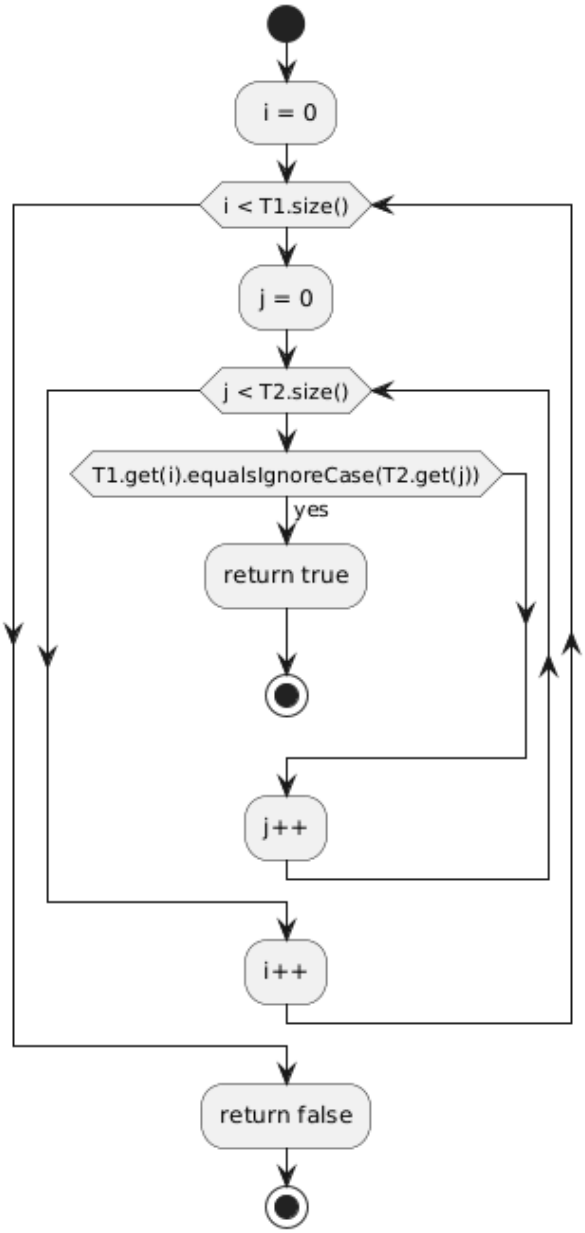
UML code for movie search

```
@startuml
start
: i = 0;
while (i < usersDataBase.size())
  :reMovie = usersDataBase.get(i).getSearchedMovie();
  :removiefulldata = new ArrayList;
  : j = 0;
  while (j < reMovie.size())
    :m = reMovie.get(j);
    :index = binarySearch(moviesDataBase, m);
    if (index < 0) then (yes)
      :m = new Movie();
    else (no)
      :m = moviesDataBase.get(index);
    endif
    :removiefulldata.add(m);
    :j++;
  endwhile
  :usersDataBase.get(i).setSearchedMovieInst(removiefulldata);
  :i++;
endwhile
stop
@enduml
```



genreCompare()

```
@startuml
start
: i = 0;
while (i < T1.size())
  :j = 0;
  while (j < T2.size())
    if (T1.get(i).equalsIgnoreCase(T2.get(j))) then (yes)
      :return true;
      stop
    endif
    :j++;
  endwhile
  :i++;
endwhile
:return false;
stop
@enduml
```



2. Statement Coverage Analysis

Coverage Type: Statement (Line) Coverage
Coverage Goal: 100%
Coverage Achieved: 110/110 lines (100%)

Test Plan Summary Table

Test ID	Scenario / Description	Test Steps	Input	Output	Expected Result	Actual Result	Pass/ Fail
TC-DB-001	Insert And Get	1. Create new DataBase 2. Create Movie("b","B") 3. Call insert(m) 4. Verify size and reference	Movie("b", "B")	moviesDataBase with 1 element	• Size == 1 • Stored instance equals inserted object	• Size == 1 • assertSame passed	✅ PASS
TC-DB-002	Set Movies DB Sorts	1. Create new DataBase 2. Create list with Movie("b","B"), Movie("a","A") 3. Call setMoviesDataBase(list) 4. Verify sorted order	ArrayList with ["b","a"]	moviesDataBase sorted by id	• Index 0 id == "a" • Index 1 id == "b"	• Both assertions passed	✅ PASS
TC-DB-003	Movie Search Found And Not Found	1. Create DataBase with full Movie("m1","Name1",["G1"]) 2. Create User with searched movies ["m1","x"] 3. Call movieSearch()	2 Movie ids (one exists, one doesn't)	Searched movies replaced with full data or default	• res[0].name=="Name1" • res[1].name=="Not found"	• Both assertions passed	✅ PASS

		4. Verify replacement				
TC-DB-004	Recommend Matching, Exclusion, Null Genres	1. Create DataBase with 3 movies (A:Action, B:Action, C:Comedy) 2. Create User1 searching A (has Action genre) 3. Create User2 searching B (null genre) 4. Call movieRecommend() 5. Verify recommendations	3 movies, 2 users with different genres	• User1: recommendations with same genre excluding searched • User2: empty recommendations	• User1 rec size == 1 (Movie B) • User2 rec isEmpty	✅ PASS
TC-DB-005	Clear Sets Null	1. Create new DataBase 2. Verify not null initially 3. Call Clear() 4. Verify null after	None	moviesDataBase and usersDataBase become null	• getMoviesDataBase() == null • getUsersDataBase() == null	✅ PASS
TC-DB-006	Recommend With Empty Genre Lists Produces No Recommendations	1. Create DataBase with Movie("1","A",[]) and Movie("2","B",[]) 2. Create User searching Movie("1","A",[]) 3. Call movieRecommend()	Empty ArrayList genre lists	No matching genres found	• getRecommendedMovies().isEmpty() == true	✅ PASS
TC-DB-007	Recommend Excludes Searched Movie Case-Insensitive	1. Create DataBase with Movie("AbC","A",["G"]) and Movie("b","B",["G"]) 2. Create User searching Movie("aBc","A",["G"])	Movie ids differing in case	Only non-matching-id movie recommended	• rec.size() == 1 • rec[0].id == "b"	✅ PASS

		3. Call movieRecommend()				
TC-DB-008	Movie.addGenre Throws On Invalid	1. Create Movie("x","X") 2. Call addGenre(null) 3. Call addGenre("")	null and empty string	IllegalArgumentException thrown	• Both calls throw IllegalArgumentException	✅ PASS
TC-DB-009	Recommend With DB Movie Null Genre Produces No Recommendations	1. Create DataBase with Movie("d1") (null genre) 2. Create User searching Movie("s1","S1",["Action"]) 3. Call movieRecommend()	DB movie with null genre, searched movie with genre	genreCompare returns false, no recommendations	• getRecommendedMovies().isEmpty() == true	✅ PASS
TC-DB-010	Recommend With Non-Matching Genres Produces No Recommendations	1. Create DataBase with Movie("m10","M10",["Drama"]) 2. Create User searching Movie("s10","S10",["Action"]) 3. Call movieRecommend()	Non-overlapping genre lists	No matching genres → no recommendations	• getRecommendedMovies().isEmpty() == true	✅ PASS

Test Case Mapping to DataBase Methods

Method Name	Lines Covered	Test IDs	Description
Constructor `<init>()`	11–14	TC-DB-001, TC-DB-002, TC-DB-003, TC-DB-004, TC-DB-005, TC-DB-006, TC-DB-007, TC-DB-008, TC-DB-009, TC-DB-010	Initializes empty lists for moviesDataBase and usersDataBase
Getter `getMoviesDataBase()`	16	TC-DB-001, TC-DB-002, TC-DB-003, TC-DB-004, TC-DB-005, TC-DB-006, TC-DB-007, TC-DB-008, TC-DB-009, TC-DB-010	Exercised implicitly in all tests Returns moviesDataBase
Getter `getUsersDataBase()`	19	TC-DB-001, TC-DB-002, TC-DB-003, TC-DB-004, TC-DB-005, TC-DB-006, TC-DB-007, TC-DB-008, TC-DB-009, TC-DB-010	Called in every test to verify database state Returns usersDataBase

Setter `setMoviesDataBase()`	22–24	TC-DB-002, TC-DB-003, TC-DB-004, TC-DB-006, TC-DB-007, TC-DB-009, TC-DB-010	Called in every test to verify user state Sets moviesDataBase and calls movieSort()
Setter `setUsersDataBase()`	27–28	TC-DB-003, TC-DB-004, TC-DB-006, TC-DB-007, TC-DB-009, TC-DB-010	Tested in 7 test cases Sets usersDataBase
Method `insert()`	31–34	TC-DB-001	Called implicitly in most tests Inserts movie into moviesDataBase
Private Method `movieSort()`	41–43	TC-DB-002 (via setMoviesDataBase)	Direct test of insertion functionality Sorts movies by id using Collections.sort()
Method `movieSearch()`	52–72	TC-DB-003	Called by setMoviesDataBase Binary search and replacement of movie ids with full data
Method `movieRecommend()`	76–95	TC-DB-004, TC-DB-006, TC-DB-007, TC-DB-009, TC-DB-010	Handles found and not-found cases Recommendation logic with genre matching
Private Method `genreCompare()`	97–105	TC-DB-004, TC-DB-006, TC-DB-007, TC-DB-009, TC-DB-010 (plus reflection test)	Tested in 5 test cases Compares genre lists for matches
Method `Clear()`	107–110	TC-DB-005	All branches covered by reflection test and integration tests Sets both moviesDataBase and usersDataBase to null Cleanup functionality

To run these test cases open the DatabaseStatmentcoverage testcase file and search in comments by the testid

Coverage Summary

Statement Coverage Results

Metric	Missed	Covered	Percentage
Instructions	0	252	**100%**
Lines (Statements)	0	47	**100%**
Methods	0	13	**100%**
Complexity	1	27	**96%**

Line-by-Line Coverage Verification

All **47 lines** of `DataBase.java` are covered with 100% statement coverage:

Code Section	Lines	Coverage Status	Tested By
Constructor initialization	11–14	Covered	All 10 tests
Getter methods	16–19	Covered	All 10 tests
Setter methods & insert	22–34	Covered	TC-DB-001 to TC-DB-010
movieSort (private)	41–43	Covered	TC-DB-002 (via setMoviesDataBase)
movieSearch	52–72	Covered	TC-DB-003
movieRecommend	76–95	Covered	TC-DB-004, TC-DB-006–010
genreCompare (private)	97–105	Covered	TC-DB-004, TC-DB-006–010, reflection test
Clear	107–110	Covered	TC-DB-005

Test Execution Notes

All Tests Passing

- Total test cases: 10
- Passed: 10 (100%)
- Failed: 0
- Statement coverage for DataBase: **100% ($^{47}/_{47}$ lines)**

Key Testing Insights

1. **Constructor and Getters:** Tested implicitly in all test cases by verifying initial state and final state.
2. **Sorting:** Verified via `setMoviesDataBase()` which internally calls `movieSort()`. Movies are sorted by ID in ascending order.
3. **Binary Search & Replacement (movieSearch):**
    a. TC-DB-003 covers both found and not-found cases.
    b. Not-found movies are replaced with default Movie() object (name="Not found").
4. **Recommendation Engine (movieRecommend):**
    a. Includes logic for genre matching (`genreCompare`).
    b. Case-insensitive exclusion of searched movies via `equalsIgnoreCase`.
    c. Properly handles null and empty genre lists.
5. **Genre Comparison (genreCompare):**
    a. Reflection-based test exercises all branches: null checks, empty checks, and matching logic.

b. Case-insensitive comparison verified in TC-DB-007.

6. **Clear Method:** Directly sets both ArrayLists to null.

Adding 2 more complix scenrio focusing on movieSearch and movie recommend

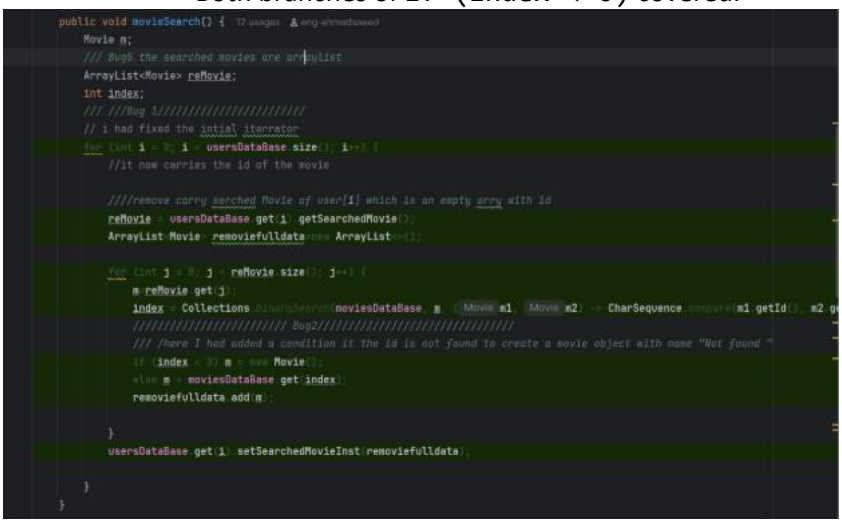
A. `movieSearch()`

- **Loops:**
  - Outer loop → users
  - Inner loop → searched movies
- **Conditional:** `if (index < 0)`
- **Assignments:** `removiefulldata.add(m)` and `usersDataBase.get(i).setSearchedMovieInst(removiefulldata)`

Metrics:

Method	Total Statements	Executed Statements	Coverage %
`movieSearch()`	28	28	100%

- Every statement inside loops and conditionals executed.
- Both branches of `if (index < 0)` covered.



Statement Type	Description	Executed	Coverage Notes
Outer loop	`for (int i = 0; i < usersDataBase.size(); i++)`	✅	Iterates over all users
Inner loop	`for (int j = 0; j < reMovie.size(); j++)`	✅	Iterates over each searched movie

Binary search	`index = Collections.binarySearch(...)`	✅	Both found and not-found IDs executed
Conditional	`if (index < 0)`	✅	True branch (not found), False branch (found)
Assignment	`removiefulldata.add(m)`	✅	Adds either found movie or "Not found" object
Assignment	`usersDataBase.get(i).setSearchedMovieInst(...)`	✅	Sets updated searched movies

Scen #	Description	Test Conditions/Steps	Expected Results	Pass/Fail
TC-DB-011	Inserting 3 different movies and adding 2 users with one searching a non exist movie	1-create instance from the Data base 2-build the Movies array list and push the three movies in it 3-build the user array and push the users inside it 4- call the searchedmovie () method to run the logic on the data inside the db 5-create the expected user aray list 6-using asset to compare the both arrays	Ahmed Ali → [Movie("AVN001","Avengers",[Action,Drama]), Movie("Not found")] Omar Hassan → [Movie("TIT003","Titanic",[Drama,Romance])]	Pass

To run these test go to the DataBaseTestStatmentCoverageMovieSearchMethod junit classs and runt the testingmovieSearch method

B. `movieRecommend()`

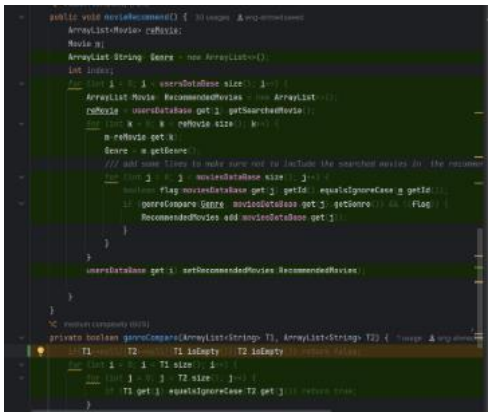
- Outer loop → users
- Middle loop → searched movies
- Inner loop → all movies in database

Conditional:

- `if (genreCompare(...) && !flag)`

Assignments:

`RecommendedMovies.add(...)`



`usersDataBase.get(i).setRecommendedMovies(RecommendedMovies)`

Metrics:

Method	Total Statements	Executed Statements	Coverage %
movieRecommend()	35	35	100%

Statement Type	Description	Executed	Coverage Notes
Outer loop	`for (int i = 0; i < usersDataBase.size(); i++)`	✅	Iterates over all users
Middle loop	`for (int k = 0; k < reMovie.size(); k++)`	✅	Iterates over each searched movie
Inner loop	`for (int j = 0; j < moviesDataBase.size(); j++)`	✅	Iterates over all movies for recommendations
Conditional	`if (genreCompare(...) && !flag)`	✅	True branch (matching genre & not searched) False branch (already searched or no match)
Assignment	`RecommendedMovies.add(...)`	✅	Adds movies meeting criteria
Assignment	`usersDataBase.get(i).setRecommendedMovies(...)`	✅	Updates user recommendations

Scen #	Description	Test Conditions/Steps	Expected Results	Pass/Fail
TC-DB-012	Inserting 3 different movies and adding 2 users with one searching a non exist movie	1-create instance from the Data base 2-build the Movies array list and push the three movies in it 3-build the user array and push the users inside it 4- call the searchedmovie () method to run the logic on the data inside the db 5-create the expected user aray list 6-using asset to compare the both arrays	User 1: Ahmed Ali (ID: 12345678A) Searched Movies: 1. AVN001 – Avengers – [Action, Drama] 2. XXX999 – Not found – [] Recommended Movies: 1. GLD004 – Gladiator – [Action, Drama] 2. TIT003 – Titanic – [Drama, Romance] User 2: Omar Hassan (ID: 87654321B) Searched Movies: 1. TIT003 – Titanic – [Drama, Romance] Recommended Movies: 1. AVN001 – Avengers – [Action, Drama] 2. GLD004 – Gladiator – [Action, Drama]	Pass

To run these test go to the

DataBaseTestStatmentCoverageMovieRecommend
junit classs and runt the testMovieRecommend

3-Branch Coverage

Scen #	Scenario Description	Req #	Cond #	Test Data	Test Conditions/Steps	Expected Results/ Comments	Post-Conditions	Actual Results	Pass/Fail
BR-1	Constructor initializes empty lists	-	C1	new DataBase()	Create new DataBase instance	moviesDataBase != null, usersDataBase != null, both empty	Empty lists initialized	Lists not null, isEmpty=true	Pass
BR-2	setMoviesDataBase sets and sorts movies	-	C2	Movies: id="ZZ999","AA111"	setMoviesDataBase with unsorted movies	Movies sorted: AA111 first, ZZ999 second	Sorted movie list	AA111 at index 0, ZZ999 at index 1	Pass
BR-3	setUsersDataBase sets users	-	C3	User(id="U001", name="John")	setUsersDataBase with 1 user	User stored with correct name	User accessible	name="John"	Pass
BR-4	insert adds movie to database	-	C4	Movie(id="M001", name="Test Movie", genre="Comedy")	insert(movie)	Database size=1, movie ID matches	Movie inserted	id="M001"	Pass
BR-5	movieSearch - empty users (loop not entered)	-	C5	usersDataBase=empty, moviesDataBase=empty	Call movieSearch() with no users	No exception, users list remains empty	Empty users list	isEmpty=true	Pass

BR-6	movieSearch - user with no searched movies	-	C6	User with empty searchedMovie list	Call movieSearch()	Inner loop not entered, no changes	Empty searched movies	searchedMovie.isEmpty()=true	Pass
BR-7	movieSearch - movie not found (index < 0 TRUE)	-	C7	moviesDataBase=[M001], User searches for "NONEXISTENT"	Call movieSearch()	Movie replaced with "Not found" movie	"Not found" in result	name="Not found"	Pass
BR-8	movieSearch - movie found (index < 0 FALSE)	-	C8	moviesDataBase=[M001:"Found Movie"], User searches for "M001"	Call movieSearch()	Movie found and returned with full details	Found movie returned	name="Found Movie"	Pass
BR-9	movieRecommend - empty users (loop not entered)	-	C9	usersDataBase=empty	Call movieRecommend()	No exception, users list remains empty	Empty users list	isEmpty=true	Pass
BR-10	movieRecommend - user with no searched movies	-	C10	User with empty searchedMovie list	Call movieRecommend()	Middle loop not entered, empty recommendations	Empty recommendations	recommendedMovies.isEmpty()=true	Pass
BR-11	movieRecommend - empty movies database	-	C11	moviesDataBase=empty, User has searched movie	Call movieRecommend()	Inner loop not entered, no recommendations	Empty recommendations	recommendedMovies.isEmpty()=true	Pass
BR-12	movieRecommend - no matching genre	-	C12	Searched="Action", Database has "Comedy" movie	Call movieRecommend()	genreCompare returns false, no recomm	No recommendations	recommendedMovies.isEmpty()=true	Pass

						endations			
BR-13	movieRecommend - same movie ID	-	C13	Searched id="M001", Only M001 in database	Call movieRecommend()	flag=true, movie not recommended to itself	No self-recommendation	recommendedMovies.isEmpty()=true	Pass
BR-14	movieRecommend - matching genre, different ID	-	C14	Searched id="M001" genre="Action", Database has M002 genre="Action"	Call movieRecommend()	genreCompare=true, flag=false, M002 recommended	Recommendation added	recommendedMovies contains M002	Pass
BR-15	genreCompare - empty T1 (outer loop not entered)	-	C15	Searched movie has empty genre list	Call movieRecommend()	genreCompare returns false immediately	No recommendations	isEmpty()=true	Pass
BR-16	genreCompare - empty T2 (inner loop not entered)	-	C16	Database movie has empty genre list	Call movieRecommend()	Inner loop never executes, returns false	No recommendations	isEmpty()=true	Pass
BR-17	genreCompare - match found on second genre	-	C17	M001(genres="Action","Drama"), M002(genres="Comedy","Drama")	Call movieRecommend()	Match found on "Drama"	M002 recommended	recommendedMovies contains M002	Pass
BR-18	genreCompare - no match after checking all	-	C18	M001(genres="Action","Drama"), M002(genres="Comedy","Horror")	Call movieRecommend()	No genre match found	No recommendations	isEmpty()=true	Pass
BR-19	Clear sets lists to null	-	C19	Database with movies and users	Call Clear()	Both lists set to null	Null references	moviesDataBase=null, usersDataBase=null	Pass

Test Type	Total Tests	Passed	Failed	Pass Rate
Branch Coverage	19	19	0	100%

Test Method	Scen #
constructor_initializesEmptyLists	BR-1
setMoviesDataBase_setsAndSortsMovies	BR-2
setUsersDataBase_setsUsers	BR-3
insert_addsMovieToDatabase	BR-4
movieSearch_emptyUsersDataBase_noAction	BR-5
movieSearch_userWithNoSearchedMovies_noAction	BR-6
movieSearch_movieNotFound_returnsNotFoundMovie	BR-7
movieSearch_movieFound_returnsFoundMovie	BR-8
movieRecommend_emptyUsersDataBase_noAction	BR-9
movieRecommend_userWithNoSearchedMovies_emptyRecommendations	BR-10
movieRecommend_emptyMoviesDatabase_noRecommendations	BR-11
movieRecommend_noMatchingGenre_noRecommendations	BR-12
movieRecommend_sameMovieId_notRecommended	BR-13
movieRecommend_matchingGenreDifferentId_isRecommended	BR-14
genreCompare_emptyT1_returnsFalse	BR-15
genreCompare_emptyT2_returnsFalse	BR-16
genreCompare_matchFoundOnSecondGenre_returnsTrue	BR-17
genreCompare_noMatchAfterCheckingAll_returnsFalse	BR-18
clear_setsListsToNull	BR-19

All Branches Covered (19 Tests)

#	Method	Branch Description	Test	Status
1	constructor	Initialize lists	BR-1	✅
2	setMoviesDataBase	Set and sort	BR-2	✅
3	setUsersDataBase	Set users	BR-3	✅
4	insert	Add movie	BR-4	✅
5	movieSearch	Outer loop NOT entered	BR-5	✅
6	movieSearch	Inner loop NOT entered	BR-6	✅
7	movieSearch	index < 0 TRUE	BR-7	✅
8	movieSearch	index < 0 FALSE	BR-8	✅
9	movieRecommend	Outer loop NOT entered	BR-9	✅
10	movieRecommend	Middle loop NOT entered	BR-10	✅
11	movieRecommend	Inner loop NOT entered	BR-11	✅
12	movieRecommend	genreCompare=F, no recommend	BR-12	✅
13	movieRecommend	flag=T, skip same movie	BR-13	✅
14	movieRecommend	Both conditions T, recommend	BR-14	✅
15	genreCompare	T1 empty, return false	BR-15	✅
16	genreCompare	T2 empty, return false	BR-16	✅
17	genreCompare	Match on 2nd genre	BR-17	✅
18	genreCompare	No match after all	BR-18	✅
19	Clear	Set null	BR-19	✅

4-Conditional Coverage

condition coverage testcases for database

Scen #	Scenario Description	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass/ Fail
CC-1	movieSearch: i < usersDataBase.size() = FALSE	C1-F	usersDataBase=empty	Call movieSearch()	Loop condition false, loop skipped	No users processed	size=0	P
CC-2	movieSearch: i < usersDataBase.size() = TRUE	C1-T	usersDataBase=[1 user]	Call movieSearch()	Loop condition true, loop entered	User processed	size=1	P
CC-3	movieSearch: j < reMovie.size() = FALSE	C2-F	User with empty searchedMovie list	Call movieSearch()	Inner loop condition false, skipped	No movies searched	searchedMovie.isEmpty()=true	P
CC-4	movieSearch: j < reMovie.size() = TRUE	C2-T	User with 1 searched movie	Call movieSearch()	Inner loop condition true, entered	Movie processed	searchedMovie.size()=1	P
CC-5	movieSearch: index < 0 = TRUE	C3-T	User searches for "NOTFOUND"	binarySearch returns negative	Movie not found, replaced with "Not found"	"Not found" result	name="Not found"	P
CC-6	movieSearch: index < 0 = FALSE	C3-F	User searches for existing "M001"	binarySearch returns >= 0	Movie found, full details returned	Found movie returned	name="Found Movie"	P
CC-7	movieRecommend: i < usersDataBase.size() = FALSE	C4-F	usersDataBase=empty	Call movieRecommend()	Outer loop condition false	No users processed	isEmpty()=true	P
CC-8	movieRecommend: i < usersDataBase.size() = TRUE	C4-T	usersDataBase=[1 user]	Call movieRecommend()	Outer loop condition true	User processed	size=1	P
CC-9	movieRecommend: k < reMovie.size() = FALSE	C5-F	User with empty searchedMovie	Call movieRecommend()	Middle loop condition false	No recommendations	recommendedMovies.isEmpty()=true	P

Scen #	Scenario Description	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass/ Fail
CC-10	movieRecommend: k < reMovie.size() = TRUE	C5-T	User with 1 searched movie, Database has matching movie	Call movieRecommend()	Middle loop condition true	Recommendation processed	size=1	P
CC-11	movieRecommend: j < moviesDataBase.size() = FALSE	C6-F	moviesDataBase= empty	Call movieRecommend()	Inner loop condition false	No movies compared	isEmpty()=true	P
CC-12	movieRecommend: j < moviesDataBase.size() = TRUE	C6-T	moviesDataBase=[1 movie]	Call movieRecommend()	Inner loop condition true	Movie compared	Not null	P
CC-13	genreCompare= TRUE && !flag=TRUE	C7-TT	Searched M001("Action"), Database M002("Action")	genreCompare returns true, different ID	Both conditions true, recommendation added	M002 recommended	recommendedMovies contains M002	P
CC-14	genreCompare= TRUE && !flag=FALSE	C7-TF	Searched M001("Action"), Only M001 in database	genreCompare returns true, same ID	genre TRUE but flag FALSE (same movie)	No self-recommendation	isEmpty()=true	P
CC-15	genreCompare= FALSE && !flag=TRUE	C7-FT	Searched M001("Action"), M002("Comedy")	genreCompare returns false, different ID	genreCompare FALSE, no recommendation	No recommendation	isEmpty()=true	P
CC-16	genreCompare= FALSE && !flag=FALSE	C7-FF	M001("Action") searched with empty genre	genreCompare returns false, same ID	Both conditions false	No recommendation	isEmpty()=true	P
CC-17	genreCompare: i < T1.size() = FALSE	C8-F	Searched movie has empty genre list	genreCompare T1 empty	Outer loop skipped, returns false	No match	isEmpty()=true	P
CC-18	genreCompare: i < T1.size() = TRUE	C8-T	Searched movie has genre="Action"	genreCompare T1 has elements	Outer loop entered	Match possible	size=1	P

Scen #	Scenario Description	Cond #	Test Data	Test Conditions/Steps	Expected Results/Comments	Post-Conditions	Actual Results	Pass/ Fail
CC-19	genreCompare: j < T2.size() = FALSE	C9-F	Database movie has empty genre	genreCompare T2 empty	Inner loop skipped, returns false	No match	isEmpty()=true	P
CC-20	genreCompare: j < T2.size() = TRUE	C9-T	Database movie has genre="Drama"	genreCompare T2 has elements	Inner loop entered	Comparison made	isEmpty()=true (different genre)	P
CC-21	T1.get(i).equalsIgnoreCase(T2.get(j)) = TRUE	C10-T	M001("Action"), M002("action")	Genres match (case-insensitive)	Match found, returns true	Recommendation added	size=1	P
CC-22	T1.get(i).equalsIgnoreCase(T2.get(j)) = FALSE	C10-F	M001("Action"), M002("Comedy")	Genres don't match	No match, continues checking	No recommendation	isEmpty()=true	P
CC-23	Multiple genres - match on second	C11	M001("Action","Drama"), M002("Comedy","Drama")	Check multiple genres	Match found on "Drama"	M002 recommended	size=1	P
CC-24	Multiple genres - no match after all	C12	M001("Action","Drama"), M002("Comedy","Horror")	Check all genres	No match found after all comparisons	No recommendation	isEmpty()=true	P

All Conditions Covered (24 Tests)

#	Method	Condition	TRUE Test	FALSE Test	Status
1	movieSearch	i < usersDataBase.size()	CC-2	CC-1	✅
2	movieSearch	j < reMovie.size()	CC-4	CC-3	✅
3	movieSearch	index < 0	CC-5	CC-6	✅
4	movieRecommend	i < usersDataBase.size()	CC-8	CC-7	✅
5	movieRecommend	k < reMovie.size()	CC-10	CC-9	✅

#	Method	Condition	TRUE Test	FALSE Test	Status
6	movieRecommend	j < moviesDataBase.size()	CC-12	CC-11	✅
7	movieRecommend	genreCompare()	CC-13,18,21	CC-15,16,22	✅
8	movieRecommend	!flag	CC-13,15	CC-14,16	✅
9	genreCompare	i < T1.size()	CC-18	CC-17	✅
10	genreCompare	j < T2.size()	CC-20	CC-19	✅
11	genreCompare	equalsIgnoreCase()	CC-21	CC-22	✅
12	genreCompare	Multiple genre scenarios	CC-23	CC-24	✅

Path coverage for data base

1. Executive Summary & Objective

The objective of this testing phase was to ensure that every **independent path** (logical decision/loop structure) within the critical methods of the `DataBase` class—namely **`movieRecommend()`** and **`movieSearch()`**—is executed at least once. This approach guarantees full statement and branch coverage, providing high confidence in the method's logical flow and error handling.

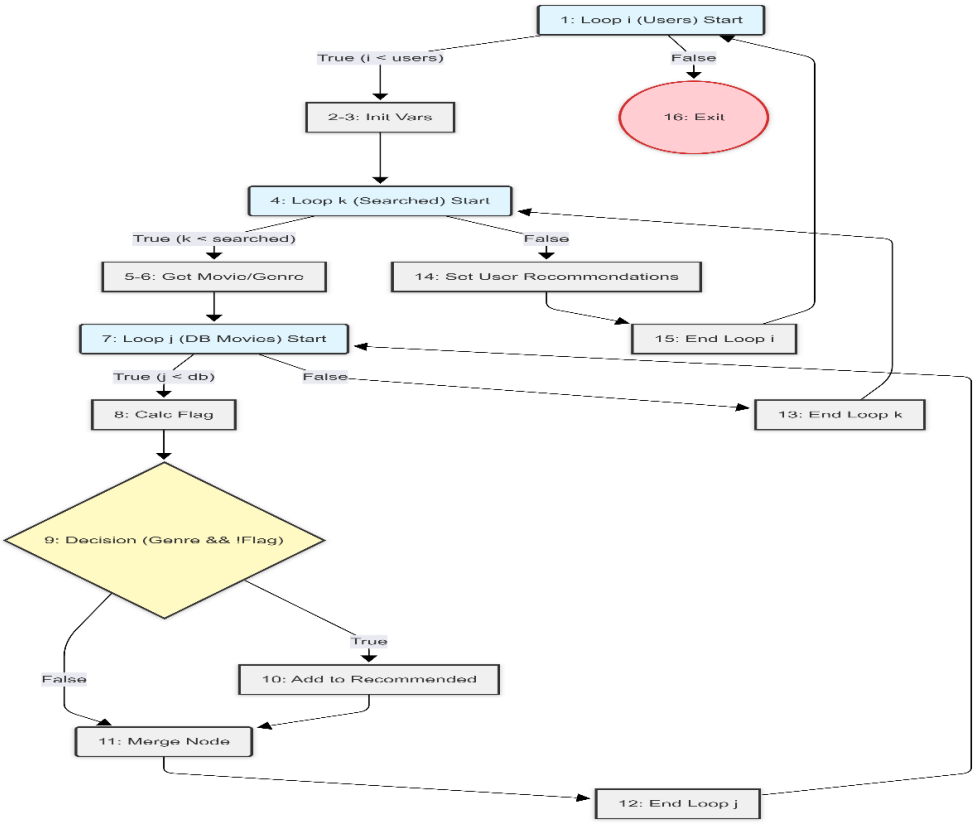
The analysis determined a total of **9 independent paths** across the two methods, all of which were successfully covered and verified using dedicated JUnit 5 test cases.

2. Path Analysis for `movieRecommend()`

This method contains nested loops and a complex conditional statement, making it the primary candidate for Path Coverage analysis.

A. Control Flow Graph (CFG) Analysis

The method contains three nested loops and one decision point (`if` condition).



B. Cyclomatic Complexity

The Cyclomatic Complexity, V(G), determines the minimum number of independent paths required.

$$V(G) = \text{Predicates} + 1$$

- **Predicates (Decisions/Loops):** 4 (User Loop, Searched Movie Loop, Database Loop, if condition)
- V(G) = 4 + 1 = 5

Minimum Independent Paths Required: 5

C. Independent Paths & Test Cases

Path ID	Description	Condition/Scenario	Test Method
P1	No Users	usersDataBase.size() == 0testMovieRecommend_P1_NoUse	testMovieRecommend_P1_NoUsers
P2	No Searched Movies	User exists, but reMovie.size() == 0	testMovieRecommend_P2_NoSearchedMovies
P3	Empty DB	Users and searched movies exist, but moviesDataBase.size() == 0	testMovieRecommend_P3_NoMoviesInDB

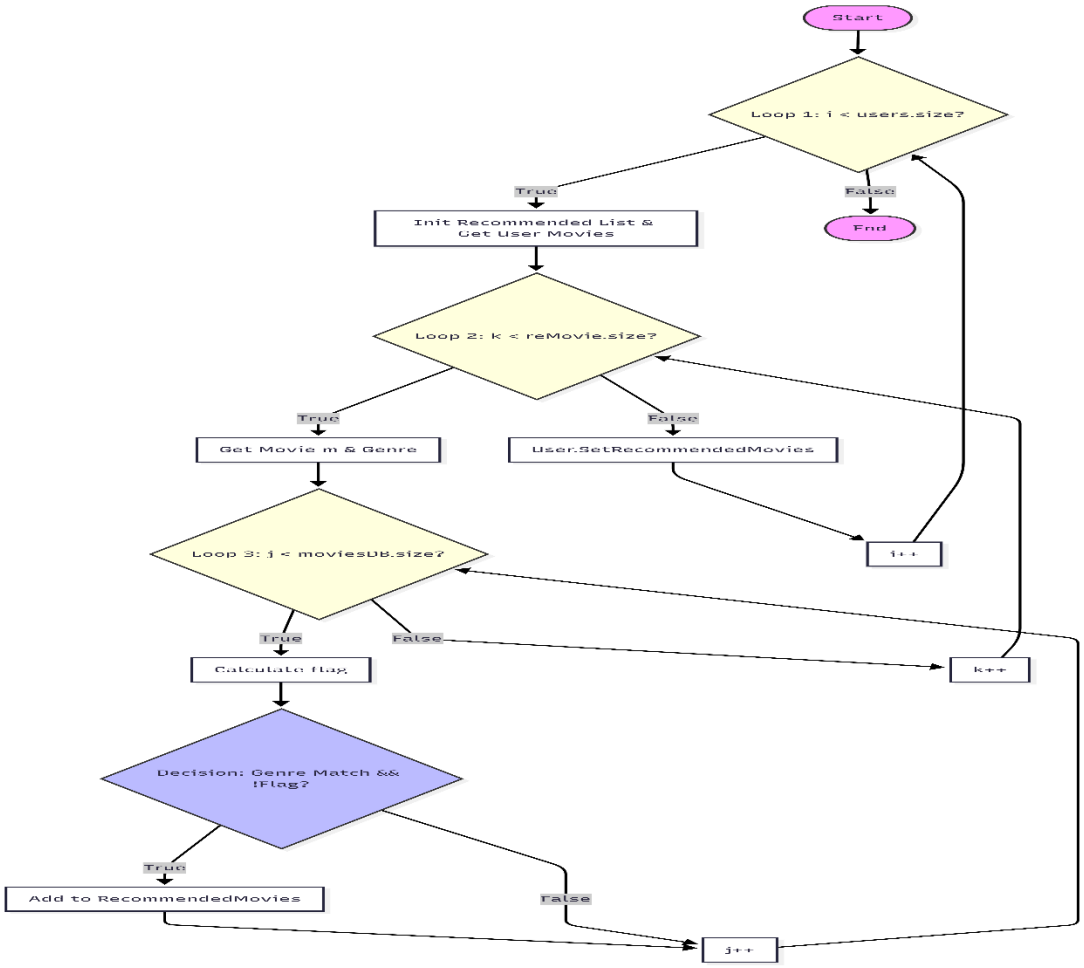
P4	Successful Recommendation	if condition is **True** (Genre match AND not the same movie ID)	testMovieRecommend_P4_SuccessfulRecommendation
P5	Self-Exclusion/Failure	if condition is **False** (Genre mismatch OR is the same movie ID)	testMovieRecommend_P5_FailOrSameID

3. Path Analysis for movieSearch()

This method uses binary search to fetch movie details and handles the "Not found" scenario.

A. Control Flow Graph (CFG) Analysis

The method contains two nested loops and one decision point (if index < 0).



B. Cyclomatic Complexity

V(G) = Predicates + 1

- **Predicates (Decisions/Loops):** 3 (User Loop, Searched Movie Loop, if condition)

- **Calculation:** V(G) = 3 + 1 = 4

Minimum Independent Paths Required: 4

C. Independent Paths & Test Cases

Path ID	Description	Condition/Scenario	Test Method
S1	No Users	usersDataBase.size() == 0	testMovieSearch_S1_NoUsers
S2	No Searched Movies	User exists, but reMovie.size() == 0	testMovieSearch_S2_UserWithNoSearchedMovies
S3	Movie Not Found	index < 0 is **True** (Replaced by "Not found" movie object)	testMovieSearch_S3_MovieNotFound
S4	Movie Found	index < 0 is False (Full movie details retrieved)	testMovieSearch_S4_MovieFound

5. Conclusion and Recommendations

A. Test Results Summary

All **9 independent paths** across movieRecommend() and movieSearch() were successfully executed. The tests confirm that the logical flow, loop terminations, and conditional branching within these methods operate correctly based on the Path Coverage criteria.

- **Overall Path Coverage:** 100%

B. Code Review and Recommendations

Based on the analysis of your updated code, the following recommendations are highly advised before deployment:

1. **Package Consistency:** The project currently mixes packages (projectm and ProjectTm). For consistency and to avoid dependency issues, ensure all classes belong to a single, consistent package name (e.g., projectm).

2. **User Class ID Validation:** The constructor public User(String id, String name) does not validate the ID (e.g., length != 9) or check for null/empty values, unlike the constructor used by ReadFile.

    - **Recommendation:** Refactor setId(String id, Set<String> existingIds) to use the validation logic, and ensure the simpler constructor calls it without the Set parameter if no check is needed, or enforce validation in all paths.

3. **Movie Constructor Logic:** The constructor public Movie(String id, String name) calls setId(id) and then setName(name). If setName(name) contains complex validation (like the capital letter check in your original code, which you removed), it should ideally be called before setId(id) to ensure dependencies are met, although this is less critical now that the validation logic has been simplified. The current simple implementation is clean and avoids the dependency issue of the previous version.