

COM1028 | Software Engineering Report

Introduction

The purpose of this report is to outline the key details involved in the technical development of Films4You's application by aiming to describe the implementation, modeling and analysis used in the process of meeting the stakeholder's expectations.

The project aims to develop a program that allows Films4You to track analytics based on data stored within their own internal database, which in return will help the company to make informed business decisions to remain above their competitors.

In order to show that requirements have been met, the report will document key diagrams and modeling and testing techniques used to aid the development of the application. By the end of the report, it hopes to analyze how the project went and how it could hope to look in the future by talking about topics such as code maintainability, system architecture and legal and ethical issues that may impact the product after its creation.

Q1 | Requirements Engineering and Their Implementation

A key aspect was for me to gauge an understanding of my potential users to design features that they are likely to find useful. Thus, I was required to build model stakeholders in order to approach the first stage of planning. I approached this by creating two distinct proto-personas:

Mia, age 33, is the regional manager for the San Francisco Bay Area Films4You stores. She is responsible for having direct oversight on operations within these stores and primarily ensures that stores are maintaining their profit margins well. She moved to California in 2016 and started work immediately there after finishing her undergraduate degree at the University of Washington, obtaining a degree in Economics.

Due to Mia's extensive background in such a course, she was promoted to a senior post after 2 years to aid the company with their finances. With such knowledge, Mia is competent in using the database system to analyse profits and compare stores under her supervision based on statistical data.

Figure 1: Proto-Persona #1

Thomas, age 22, is an IT technician based within the Downtown Manhattan Films4You store in New York. Thomas is responsible for regularly maintaining both electronic and software systems for the company while being able to keep data stored on such systems up-to-date. He initially moved to the company after gaining his college diploma in IT Systems. Thomas worked for the company under an internship role in 2018 and soon after, was established as a full-time employee after graduation.

Thomas wishes to one day be able to use his skills and knowledge acquired at the company to start his own business. Inspired with the advancement of AI technology, he wishes to implement this to create software that will allow for maintenance of computer systems to be conducted with the help of artificial programs.

Figure 2: Proto-Persona #2

By doing this it aided me in envisaging possible difficulties that they might have in understanding and using product features while also identifying the type of skill sets and characteristics my stakeholder could potentially have.

It is important for me to understand the implications of my system on the user and how they are intended to use it. Therefore, it is crucial to build a bigger picture on how the software I am designing is to be used in day to day activities for employees at the company:

As a manager, I want to be able to use the software to help me generate a report so that I can present statistical evidence and my hypothesis to executives so they can make informed decisions about the company.

Figure 3: Mia's User-Story

As an employee, I want to be able to use the software to help me with managing the database so that I can add or remove elements with ease.

Figure 4: Thomas' User-Story

By depicting an example stakeholder utilizing my product's features, it aids me during the design process:

Mia has been asked by the Chief Financial Officer of Films4You, Leo, to help establish evidence on whether or not the number of rentals of a movie is proportional to the number of popular actors cast in the films, within her area of oversight. Leo wants Mia to plot graphs and collect data stored within the company's database systems in order for executive members at the company to analyse.

Mia starts by logging into the interface in order to establish a dataset to model the scenario. She does this by using the system to find the top 500 most popular films based on number of rentals and the top 100 most popular actors based on the number of appearances in movies. Mia uses both data sets to find a correlation between the two by plotting graphs and using statistical analysis to back up her hypothesis.

Once her hypothesis has been proved with substantial evidence, Mia logs back in to the system to store and publish her findings. Mia logs out to ensure no unauthorized access is obtained. This allows for the executive officers at the company to log in to the system as an administrator so that such information can be collected and brought forward during a meeting.

Figure 5: Mia's User-Scenario

My first requirement was to **find the total number of actors**. Using the test database, it was necessary to validate my solutions. Within my Java application, requirement 1 was all fulfilled in a single class that contained methods to retrieve information from the database. In this instance, totalling the row count of the queried search by utilizing the global objects declared at the beginning of the test class, and comparing it to the real value was my strategy.

```
Requirement r = new Requirement();
Database db = new Database();
ResultSet queryResult = db.query("SELECT COUNT(actor_id) AS rowcount FROM actor");
```

Figure 6: Initializing Global Variables

```
@Test
public void testGetTotalActors() {
    try {
        queryResult.next();
        String rowCount = queryResult.getString("rowcount");
        System.out.println("The number of actors is " + rowCount + ".");
        assertEquals(rowCount, r.getValueAsString());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve total number of actors.");
    }
}
```

Figure 7: getTotalActors() Test Method

The last step involved translating the first test method into a more readable format so that the user can understand what is taking place, while displaying the correct amount of actors:

```
@Test
public void testGetValueAsString() {
    try {
        assertEquals(queryResult.getString("rowcount"), r.getValueAsString());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve value.");
    }
}

@Test
public void testGetHumanReadable() {
    try {
        assertEquals("The total number of actors is " + queryResult.getString("rowcount") + ".",
            r.getHumanReadable());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail();
    }
}
```

Figure 8: getValueAsString() and getHumanReadable() Test Methods

Q2 | Modeling and Programming Using UML Class Diagrams

My second task required me to **find the top 10 most popular films based on the number of rentals**. It was important that I first modeled my requirement in a UML class diagram to display the relationships between the objects and methods within my system and to allow for a sense of orientation. A 1 to 1 relationship was present between the Requirement class and the Database in order to establish a connection, and implements a RequirementInterface with mandatory methods.

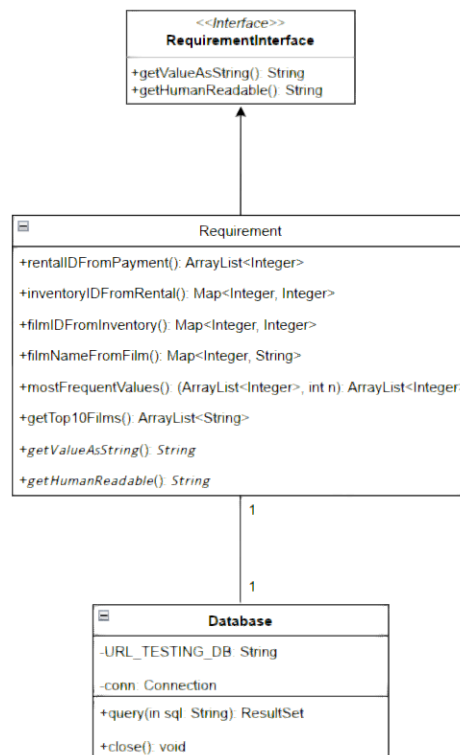


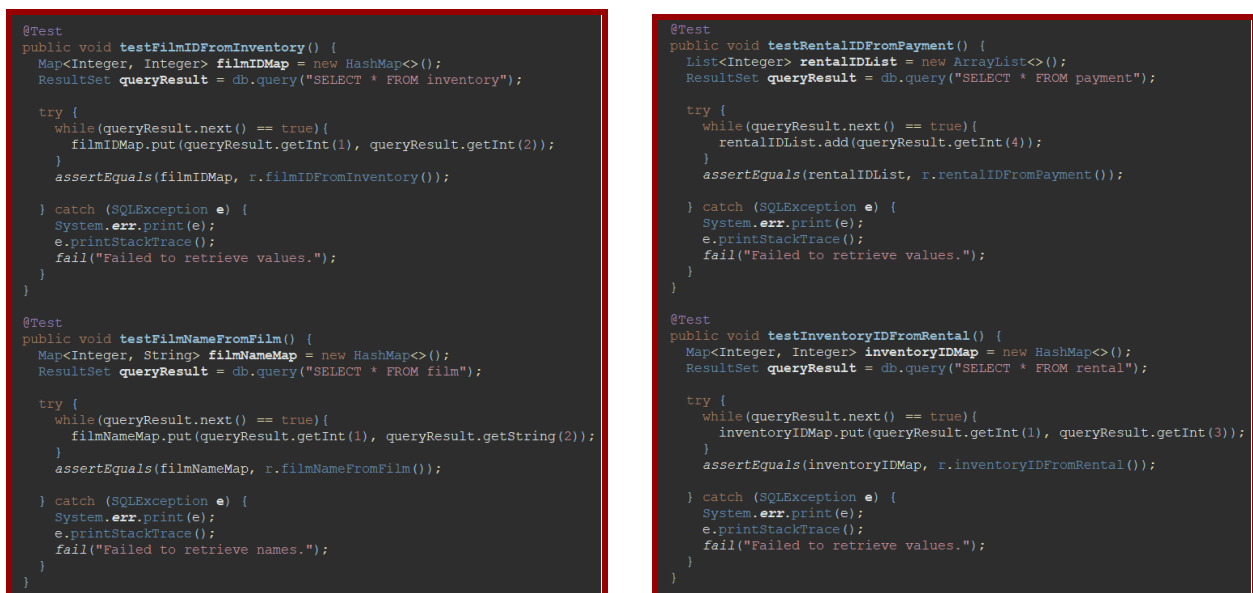
Figure 9: Requirement 2 UML Class Diagram

I decided to model my requirements with a one class based design for a few reasons:

- A single class with methods only is simpler to understand and maintain than many classes. It reduces the complexity of my code by keeping all the related methods in one place.
- My tasks were all similar in nature. It was easier to reuse my code for the rest of my requirements, allowing the software to be constructed in a faster manner.

- It adheres to the idea of encapsulation, as it keeps it separate from other parts of my application, making my code more modular and easier to maintain over time.
- Allows for easier testing. This way, I created a test class that tests all the methods in the main class.

The test code uses database queries to retrieve the necessary data for comparison with the expected results. If any errors occur during the execution of the tests, the test will fail and an error message will be printed. More importantly, it includes methods to retrieve rental IDs from payment records, inventory IDs from rental records, film IDs from inventory records in order to fetch the corresponding film names from film records by designating keys and values to place the database elements in either a hashmap or arraylist:



```

@Test
public void testFilmIDFromInventory() {
    Map<Integer, Integer> filmIDMap = new HashMap<>();
    ResultSet queryResult = db.query("SELECT * FROM inventory");

    try {
        while(queryResult.next() == true){
            filmIDMap.put(queryResult.getInt(1), queryResult.getInt(2));
        }
        assertEquals(filmIDMap, r.filmIDFromInventory());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve values.");
    }
}

@Test
public void testFilmNameFromFilm() {
    Map<Integer, String> filmNameMap = new HashMap<>();
    ResultSet queryResult = db.query("SELECT * FROM film");

    try {
        while(queryResult.next() == true){
            filmNameMap.put(queryResult.getInt(1), queryResult.getString(2));
        }
        assertEquals(filmNameMap, r.filmNameFromFilm());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve names.");
    }
}

@Test
public void testRentalIDFromPayment() {
    List<Integer> rentalIDList = new ArrayList<>();
    ResultSet queryResult = db.query("SELECT * FROM payment");

    try {
        while(queryResult.next() == true){
            rentalIDList.add(queryResult.getInt(4));
        }
        assertEquals(rentalIDList, r.rentalIDFromPayment());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve values.");
    }
}

@Test
public void testInventoryIDFromRental() {
    Map<Integer, Integer> inventoryIDMap = new HashMap<>();
    ResultSet queryResult = db.query("SELECT * FROM rental");

    try {
        while(queryResult.next() == true){
            inventoryIDMap.put(queryResult.getInt(1), queryResult.getInt(3));
        }
        assertEquals(inventoryIDMap, r.inventoryIDFromRental());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve values.");
    }
}

```

Figure 10 and 11: testRentalIDFromPayment and testInventoryIDFromRental() and testFilmIDFromInventory() and testFilmNameFromFilm() Test Methods

I had to also construct a test to determine whether or not the system can correctly identify the most frequently occurring values in a list and return the top 10 most popular films based on rental records. An expected value was compared against the actual value to test whether or not the method would return the correct array in order.

```

@Test
public void testMostFrequentValues() {
    ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 2, 3, 3, 4, 4, 4, 4));
    ArrayList<Integer> expectedValues = new ArrayList<>(Arrays.asList(4, 3, 2, 1));

    ArrayList<Integer> actualValues = Requirement.mostFrequentValues(list, 4);

    assertEquals(expectedValues, actualValues);
}

```

Figure 12: testMostFrequentValues() Test Method

The system needed to also be able to correctly retrieve a list of films and return the top 10 most popular films based on rental records. By first querying the actual result expected from the test database, I could compare it to what my method returned.

```

@Test
public void testTop10Films() {
    ArrayList<String> actualFilms = new ArrayList<>();
    ResultSet queryResult = db.query("SELECT f.title " +
        "FROM film f " +
        "JOIN inventory i ON f.film_id = i.film_id " +
        "JOIN rental r ON i.inventory_id = r.inventory_id " +
        "JOIN payment p ON r.rental_id = p.rental_id " +
        "GROUP BY f.film_id " +
        "ORDER BY COUNT(*) DESC, f.title ASC " + "LIMIT 10");

    try {
        while (queryResult.next()) {
            String title = queryResult.getString("title");
            actualFilms.add(title);
        }
        assertEquals(actualFilms, r.getTop10Films());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve films.");
    }
}

```

Figure 13: testMostFrequentValues() Test Method

It was important that the program was capable of formatting this information into a human-readable string too, so comparisons were used to determine whether the method ran without issue:

```

@Test
public void testGetValueAsString() {
    ArrayList<String> filmsAsString = new ArrayList<>();
    ResultSet queryResult = db.query("SELECT film.title, COUNT(rental.rental_id) AS rental_count " +
        "FROM rental " +
        "JOIN inventory ON rental.inventory_id = inventory.inventory_id " +
        "JOIN film ON inventory.film_id = film.film_id " +
        "GROUP BY film.film_id " +
        "ORDER BY rental_count DESC, film.title ASC " +
        "LIMIT 10");

    try {
        while (queryResult.next()) {
            String title = queryResult.getString("title");
            filmsAsString.add(title);
        }
        assertEquals(filmsAsString.toString(), r.getValueAsString());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve films.");
    }
}

```

```

@Test
public void testGetHumanReadable() {
    ArrayList<String> getTop10Films = new ArrayList<>();
    ResultSet queryResult = db.query("SELECT film.title, COUNT(rental.rental_id) AS rental_count " +
        "FROM rental " +
        "JOIN inventory ON rental.inventory_id = inventory.inventory_id " +
        "JOIN film ON inventory.film_id = film.film_id " +
        "GROUP BY film.film_id " +
        "ORDER BY rental_count DESC, film.title ASC " +
        "LIMIT 10");

    try {
        while (queryResult.next()) {
            String title = queryResult.getString("title");
            getTop10Films.add(title);
        }
        assertEquals("The top 10 most popular films based on numbers of rentals are: " + "\n 1." + getTop10Films.get(0) +
            "\n 2." + getTop10Films.get(1) + "\n 3." + getTop10Films.get(2) + "\n 4." + getTop10Films.get(3) +
            "\n 5." + getTop10Films.get(4) + "\n 6." + getTop10Films.get(5) + "\n 7." + getTop10Films.get(6) +
            "\n 8." + getTop10Films.get(7) + "\n 9." + getTop10Films.get(8) + "\n 10." + getTop10Films.get(9), r.getHumanReadable());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail();
    }
}

```

Figure 14 and 15: getValueAsString() and getHumanReadable() Test Methods

Q3 | Modeling and Programming Using UML Sequence Diagrams

My third task required me to **find all the actors with the first name 'PENELOPE'**.

Documenting the system's requirements allowed me to flush out the design of the system, to prevent redundant code and view the interactions between objects. Therefore, I constructed a UML sequence diagram to portray the different types of control structures involved:

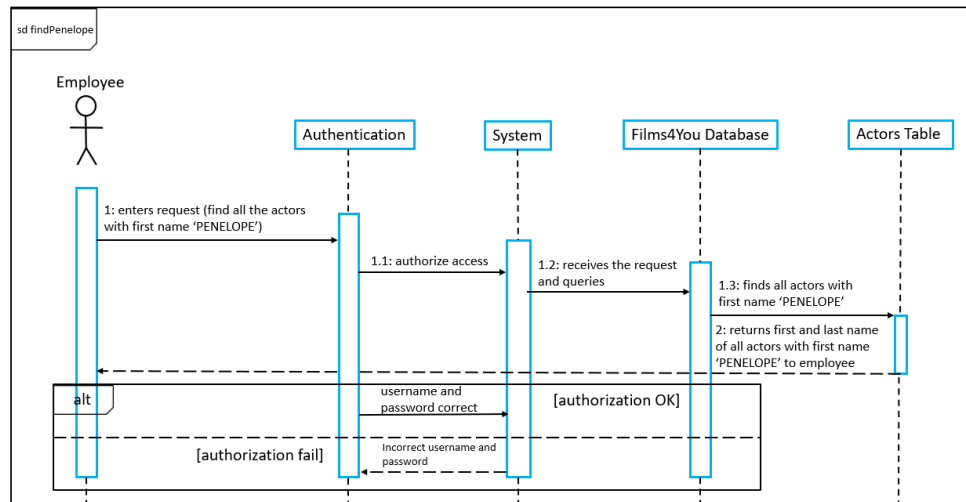


Figure 16: testMostFrequentValues() Test Method

The *Authenticator*'s purpose is to verify the user's credentials before allowing them to access the system, by using a username and password system. If the credentials are correct, authorization is granted to the system to receive and request the query. We assume that the employee has logged in the following documented code (1.1), and that a user interface has been constructed.

The *system* receives the user's request and queries the database to find all actors with the first name PENELOPE. The system also handles any errors or exceptions that may occur during the process. 1.2 describes the process of receiving and querying the request made, as shown:

```

public ArrayList<String> findPenelopes() {
    // Initialise ArrayList, Database and ResultSet objects to allow for querying.
    Database db = new Database();
    ResultSet queryResult = db.query("SELECT * FROM actor");
    ArrayList<String> listOfPenelopes = new ArrayList<>();

    try {
        // While the cursor is pointed at the first row and if "first_name" is "PENELOPE", add "last_name" to ArrayList "listOfPenelopes".
        while(queryResult.next() == true) {
            if(queryResult.getString(2).equals("PENELOPE")) {
                String lastName = queryResult.getString(3);
                listOfPenelopes.add(lastName);
            }
        }
    }
}
  
```

Figure 17: findPenelope() Method (1.2)

The *database* is designed to store information about actors, including their first names. A query is written to find all actors with the first name PENELOPE. A database query “SELECT * FROM actor” is made. While the pointer is pointed at the first row and if the first names contain “PENELOPE”, the actor’s last names are added to a list. 1.3 describes this process as shown below:

```
try {
    // While the cursor is pointed at the first row and if "first_name" is "PENELOPE", add "last_name" to ArrayList "listOfPenelopes".
    while(queryResult.next() == true) {
        if(queryResult.getString(2).equals("PENELOPE")) {
            String lastName = queryResult.getString(3);
            listOfPenelopes.add(lastName);
        }
    }
}
```

Figure 18: Database Query (1.3)

2 describes a scenario in which the first and last names of the query search are returned for the employee to see, a reference to the **getHumanReadable()** method which formats the query appropriately:

```
/**
 * Method to return the names of actors with first name "PENELOPE" in a human readable format.
 *
 * @return Name of actors with first name "PENELOPE" for a human to read.
 */
@Override
public @NonNull String getHumanReadable() {
    ArrayList<String> lastNames = findPenelopes();
    ArrayList<String> penelopeNames = new ArrayList<>();
    String γ = "PENELOPE ";
    String δ = null;

    for(String name : lastNames) {
        δ = γ.concat(name);
        penelopeNames.add(δ);
    }

    return "The following actors have the name PENELOPE: " + "\n" + penelopeNames.get(0) + "\n" +
penelopeNames.get(1) + "\n" +
penelopeNames.get(2) + "\n" +
penelopeNames.get(3) + "\n";
}
```

Figure 19: getHumanReadable() Method (2)

The method first initializes two array lists to store the lastName by recursively calling the **findPenelopes()** function and storing the first and last name of every actor with the first name “PENELOPE”. It concatenates each last name in front of a first name, as every first name will start with “PENELOPE” and returns each element in the penelopesNames array list in a formatted structure.

A test class was produced to validate each method within the UML Sequence diagram. The first method, **testFindPenelopes()** executes a SQL query to retrieve the last names of all actors with the first name 'PENELOPE'. The test method then iterates over the query result set and adds each last name to an ArrayList.

```
Requirement r = new Requirement();
Database db = new Database();
ResultSet queryResult = db.query("SELECT first_name, last_name FROM actor WHERE first_name = 'PENELOPE'");

@Test
public void testFindPenelopes() {
    ArrayList<String> penelopes = new ArrayList<>();

    try {
        while(queryResult.next() == true) {
            String lastName = queryResult.getString("last_name");

            penelopes.add(lastName);
        }
        assertEquals(penelopes, r.findPenelopes());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve names.");
    }
}
```

Figure 20: Initializing Global Variables and testFindPenelopes() Test Method
testGetValueAsString() compares array lists of full names and ensures they are returned as strings. The first and last names are both concatenated and returned by **getValueAsString()**.

```
@Test
public void testGetValueAsString() {
    ArrayList<String> penelopesAsString = new ArrayList<>();

    try {
        while(queryResult.next() == true) {
            String firstName = queryResult.getString("first_name");
            String lastName = queryResult.getString("last_name");

            penelopesAsString.add(firstName + " " + lastName);
        }
        assertEquals(penelopesAsString.toString(), r.getValueAsString());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve names.");
    }
}
```

Figure 21: testGetValueAsString Test Method

testGetHumanReadable() compares a human-readable string generated to an expected string.

```
@Test
public void testGetHumanReadable() {
    ArrayList<String> penelopeAsReadable = new ArrayList<>();

    try {
        while(queryResult.next() == true) {
            String firstName = queryResult.getString("first_name");
            String lastName = queryResult.getString("last_name");

            penelopeAsReadable.add(firstName + " " + lastName);
        }
        assertEquals("The following actors have the name PENELOPE: " + "\n" + penelopeAsReadable.get(0) + "\n" +
            penelopeAsReadable.get(1) + "\n" +
            penelopeAsReadable.get(2) + "\n" +
            penelopeAsReadable.get(3) + "\n", r.getHumanReadable());
    } catch (SQLException e) {
        System.err.print(e);
        e.printStackTrace();
        fail("Failed to retrieve names.");
    }
}
```

Figure 22: testGetHumanReadable Test Method

Q4 | Extra Skills Demonstrated in Modeling and/or Programming

My fourth requirement was to **find the most popular film in each category based on the number of rentals**. I decided to model this with the use of a UML Class diagram, in order to be able to analyze what code I could use from my previous requirements to help with the construction:

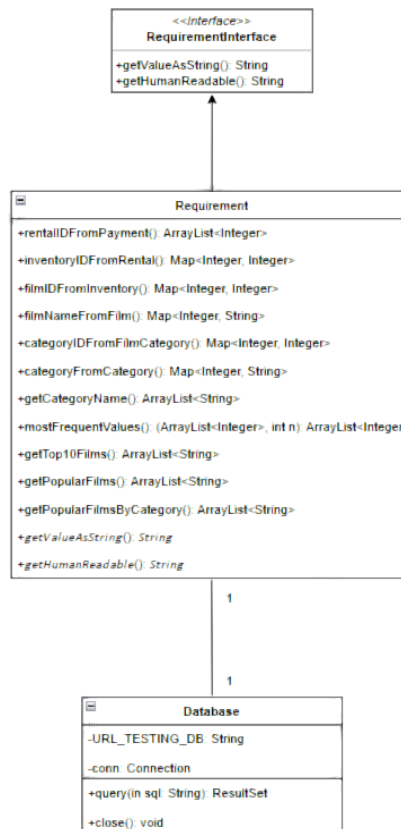


Figure 23: Requirement 4 UML Class Diagram

My one class system allowed me to save time during the project by reusing methods found in my Requirement 2. With this, it allowed me to implement 4 new methods instead. The program interacts with a database to retrieve information about films and categories

The **categoryIDFromFilmCategory()** method retrieves a mapping of film IDs to their corresponding category IDs from the film_category table in the database.

```
/**
 * Method for returning a map of all category id's within the film_category table by using the film_id as a key
 * in order to get the corresponding category id.
 *
 * @return Map of all values within the "film_id" and the corresponding "category_id" column or null on error.
 * @exception Catch SQLException in the case of error, print stack trace and return null.
 */
public Map<Integer, Integer> categoryIDFromFilmCategory(){
    // Initialise HashMap, Database and ResultSet objects to allow for querying.
    Database db = new Database();
    ResultSet queryResult = db.query("SELECT * FROM film_category");
    Map<Integer, Integer> filmCategoryMap = new HashMap<>();

    try {
        // While the cursor is pointed at the first row, place "film_id" and "category_id" into the map.
        while(queryResult.next() == true) {
            filmCategoryMap.put(queryResult.getInt(1), queryResult.getInt(2));
        }
    } catch(SQLException e) {
        e.printStackTrace();
        return null;
    }

    return filmCategoryMap;
}
```

Figure 24: categoryIDFromFilmCategory() Method

categoryFromCategory() retrieves a mapping of category IDs to their corresponding names from the category table in the database.

```
/**
 * Method for returning a map of all categories within the category table by using the category_id as a key
 * in order to get the corresponding category
 *
 * @return Map of all values within the "category_id" and the corresponding "name" column or null on error.
 * @exception Catch SQLException in the case of error, print stack trace and return null.
 */
public Map<Integer, String> categoryFromCategory(){
    // Initialise HashMap, Database and ResultSet objects to allow for querying.
    Database db = new Database();
    ResultSet queryResult = db.query("SELECT * FROM category");
    Map<Integer, String> categoryMap = new HashMap<>();

    try {
        // While the cursor is pointed at the first row, place "category_id" and "name" into the map.
        while(queryResult.next() == true) {
            categoryMap.put(queryResult.getInt(1), queryResult.getString(2));
        }
    } catch(SQLException e) {
        e.printStackTrace();
        return null;
    }

    return categoryMap;
}
```

Figure 25: categoryFromCategory() Method

The **getCategoryName()** method retrieves a list of category names from the category table in the database.

```
/**
 * Method for returning an ArrayList of all categories within the category table.
 *
 * @return ArrayList of all values within the "name" column or null on error.
 * @exception Catch SQLException in the case of error, print stack trace and return null.
 */
public ArrayList<String> getCategoryName() {
    // Initialise Database and ResultSet objects to allow for querying.
    Database db = new Database();
    ResultSet queryResult = db.query("SELECT * FROM category");
    ArrayList<String> listOfCategories = new ArrayList<>();

    try {
        // While the cursor is pointed at the first row, place "payment_id" and "rental_id" into the map.
        while(queryResult.next() == true) {
            String name = queryResult.getString(2);
            listOfCategories.add(name);
        }
    } catch(SQLException e) {
        e.printStackTrace();
        return null;
    }

    return listOfCategories;
}
```

Figure 26: getCategoryName() Method

The **getPopularFilms()** method retrieves the top 10 most popular films based on the number of rentals, by first querying the payment table to get a list of rental IDs, then using those IDs to retrieve corresponding inventory IDs from the rental table, and finally using those inventory IDs to retrieve corresponding film IDs from the inventory table. The method then uses the **mostFrequentValues()** method to determine the most frequently rented films, and returns the corresponding film titles.

```
/**
 * Method for sorting an ArrayList of integers by the most frequently recurring values in order from highest to lowest up
 * to the nth value.
 *
 * @return ArrayList of most frequently recurring values in order up to an nth entry.
 * @param ArrayList list of type Integer representing the list we wish to order by most recurring values.
 * @param int n representing the nth entry.
 */
public static ArrayList<Integer> mostFrequentValues(ArrayList<Integer> list, int n) {
    ArrayList<Integer> mostFrequentValues = new ArrayList<>();
    HashMap<Integer, Integer> frequencyMap = new HashMap<>();

    // Calculate the frequency of each value in the list by first iterating over the ArrayList.
    for (Integer value : list) {
        frequencyMap.put(value, frequencyMap.getOrDefault(value, 0) + 1);
    }

    // Sort the map by value in descending order and extract the first n entries.
    List<Map.Entry<Integer, Integer>> sortedEntries = new ArrayList<>(frequencyMap.entrySet());
    sortedEntries.sort(Map.Entry.comparingByValue(Comparator.reverseOrder()));
    for (int i = 0; i < n && i < sortedEntries.size(); i++) {
        mostFrequentValues.add(sortedEntries.get(i).getKey());
    }

    return mostFrequentValues;
}
```

Figure 27: mostFrequentValues() Method

```

/**
 * Method for returning an ArrayList of the top 10 most popular films based on rentals.
 * @return ArrayList of top 10 films.
 */
public ArrayList<String> getPopularFilms() {
    // Initialise list and maps.
    List<Integer> paymentList = rentalIDFromPayment();
    Map<Integer, Integer> rentalMap = inventoryIDFromRental();
    Map<Integer, Integer> inventoryMap = filmIDFromInventory();
    Map<Integer, String> filmMap = filmNameFromFilm();

    // Initialise an ArrayList to contain all possible film id's.
    ArrayList<Integer> filmIDList = new ArrayList<>();

    // Iterate over the list of rental id's collected from the payment table.
    for (Integer α : paymentList) {
        // Collect all rental id's from the rentalMap to get a corresponding inventory_id. Initialise this as an Integer x.
        Integer x = rentalMap.get(α);
        // Collect all inventory id's from the inventoryMap to get a corresponding film_id. Initialise this as an Integer y.
        Integer y = inventoryMap.get(x);
        // Add the corresponding film_id to an ArrayList "filmIDList".
        filmIDList.add(y);
    }

    // Initialise an ArrayList to contain the most popular films based on rentals.
    ArrayList<String> mostPopularFilms = new ArrayList<>();
    // Initialise an ArrayList to contain the most frequently recurring film id's.
    ArrayList<Integer> mostFrequentID = mostFrequentValues(filmIDList, 500);

    // Iterate over the ArrayList of most frequently recurring film id's.
    for (Integer β : mostFrequentID) {
        // Collect the top 10 most frequently recurring film id's to get a corresponding title. Initialise this as a String z.
        String z = filmMap.get(β);
        // Add the corresponding titles to an ArrayList "mostPopularFilms".
        mostPopularFilms.add(z);
    }

    return mostPopularFilms;
}

```

Figure 28: getPopularFilms() Method

getPopularFilmByCategory() returns the most popular film from each category in a string format. It initializes arraylists and a map to store film IDs and their corresponding category IDs, then iterates over the map and adds each film ID to the appropriate category list based on its category ID using a switch statement.

A new list is created to store the sorted lists of film IDs based on frequency of occurrence within each category. The **mostFrequentValues()** method is called to sort the film IDs by frequency. A second list `returnList` is created that contains the most popular film ID from each category. By iterating over this list, a map is used to retrieve the corresponding film name and add it to the final arraylist of films. The method then returns the arraylist containing the most popular film from each category.

getValueAsString() returns a string representation of the most popular film from each category by calling the **getPopularFilmByCategory()** method, which returns a list of the most popular films from each category. This list is then converted to a string using the **toString()** method and returned.

```
/**
 * Method to return the most popular film from each category as a string.
 *
 * @return Most popular film from each category.
 */
@Override
public @Nullable String getValueAsString() {
    return getPopularFilmByCategory().toString();
}
```

Figure 29: **getValueAsString()** Method

The last method, **getHumanReadable()**, returns a human-readable string representation of the most popular film from each category. It does this by calling **getPopularFilmByCategory()** and then concatenating the category names and the corresponding film names into a single string with appropriate formatting. The resulting string contains the top film in each category, presented in a clear and readable format.

```
/**
 * Method to return the most popular film from each category in a human readable format.
 *
 * @return Most popular film from each category for a human to read.
 */
@Override
public @NonNull String getHumanReadable() {
    return "The top action movie is: " + getPopularFilmByCategory().get(0) + "\n" +
        "The top animation movie is: " + getPopularFilmByCategory().get(1) + "\n" +
        "The top childrens movie is: " + getPopularFilmByCategory().get(2) + "\n" +
        "The top classics movie is: " + getPopularFilmByCategory().get(3) + "\n" +
        "The top comedy movie is: " + getPopularFilmByCategory().get(4) + "\n" +
        "The top documentary movie is: " + getPopularFilmByCategory().get(5) + "\n" +
        "The top drama movie is: " + getPopularFilmByCategory().get(6) + "\n" +
        "The top family movie is: " + getPopularFilmByCategory().get(7) + "\n" +
        "The top foreign movie is: " + getPopularFilmByCategory().get(8) + "\n" +
        "The top game movie is: " + getPopularFilmByCategory().get(9) + "\n" +
        "The top horror movie is: " + getPopularFilmByCategory().get(10) + "\n" +
        "The top music movie is: " + getPopularFilmByCategory().get(11) + "\n" +
        "The top new movie is: " + getPopularFilmByCategory().get(12) + "\n" +
        "The top action sci-fi is: " + getPopularFilmByCategory().get(13) + "\n" +
        "The top sports movie is: " + getPopularFilmByCategory().get(14) + "\n" +
        "The top travel movie is: " + getPopularFilmByCategory().get(15) + "\n";
}
```

Figure 29: **getHumanReadable** Method

Q5 | Critical Analysis and Reflection

My final product works as intended and has met all requirement criterias. Not only is the program functional in all aspects, it was completed in a timely manner. I managed to successfully frame all requirements appropriately by creating models to describe each process within the system as best as possible, and demonstrate all relationships involved. Thus, it allowed me to communicate to my stakeholders on what is being produced and help me make informed decisions on what is expected of my product.

I had decided to implement the majority of my requirements using a one class based system. If I was to go back and improve, I would have opted for a system that utilizes multiple classes.. My main thought process behind this is scalability, as doing it this way would allow for code implementation in the future to be a much simpler process. In addition, it would offer better abstraction, the ability to represent complex systems using simpler, more general concepts. This would have reduced the complexity of my program, making it easier to understand. An improvement to my model would have been to add more aspects of inheritance, a simpler method of reusing properties and methods to make my system more defined and extend the functionality of classes, while being easier to follow.

It is important that I take into account future considerations for the system architecture that I have developed by designing the structure and components of the system in such a way that it can meet the current and future requirements of the company.

One possible system architecture for Films4You could be a three-tier architecture consisting of a presentation layer, application layer, and data layer. It would be responsible for providing a user-friendly interface for staff and customers to interact with the system via a web-based application, integration of the business logic of the system, where the requests from the presentation layer would be processed and the already constructed database that stores all the necessary data related to the system utilizing SQLite. My intentions behind using this architecture are simple. It provides a clear separation of concerns, making the system easier to develop, maintain, and scale as it minimizes complexity.

The software would be required to adhere to legal laws and take into account any potential ethical issues. Films4You collects sensitive personal information from its customers, such as their name, address, and payment information. These pieces of information could be stored in separate database servers to prevent an attacker from obtaining all credentials. so it is important to ensure that this information is stored and handled securely to prevent unauthorized access, theft, or misuse. Deploying data encryption protocols, using database and web application firewalls and using methods of physical database security such as locks and security staff at the database center will all contribute to prevention of leaked personal information.

The system should be designed to comply with relevant data privacy and security laws. For Films4You stores located in regions within the European Union, the General Data Protection Regulation (GDPR) should be followed.

Stores located in the UK should be required to follow the Data Protection Act (2018) and stores in the US would adhere to the California Consumer Privacy Act (CCPA). This includes implementing appropriate security measures such as encryption of sensitive data, two-factor authentication for staff access, and regular backups of the database to prevent data loss.

For stronger relations between the company and customers, Films4You should set out to obtain explicit consent from customers before collecting and storing their personal information, and to provide clear and transparent information on how their data will be used and protected in order to stay within regulations that deal with freedom of information. Failure to comply with data privacy and security laws can result in legal action, fines, and damage to the reputation of the company. Therefore, it is crucial for Films4You to prioritize data privacy and security in the design and development of its system.

These are some of the many ways that professional aspects of software engineering can be implemented while developing the system and can help Films4You make informed business decisions and remain competitive.