

IMDB-app (fase 3)

Memoria del proyecto.

Aingeru García
Alba Gonzalez Zarpón
Sergio de los Toyos Sandoval

Índice

1. Introducción	2
2. Fase 1: Estructuras lineales y búsqueda dicotómica	2
2.1. Estructura general	2
2.1.1. Sobre la ley de Demeter	4
2.1.2. Diagrama UML	5
2.2. Análisis de algoritmos y métodos.	6
2.2.1. Búsqueda	6
2.2.2. Carga de datos	7
2.3. Ordenación	9
2.3.1. Análisis (Mergesort iterativo)	9
2.4. Simulación (tiempos)	10
2.5. Casos de prueba	11
2.6. Observaciones y comentarios	11
3. Fase 2: Árboles binarios de búsqueda	12
3.1. Cambios con respecto a la fase 1	12
3.1.1. Changelog	12
3.1.2. Wrappers	12
3.2. Estructura general	14
3.2.1. Diagrama UML	15
3.3. Análisis de algoritmos y métodos.	15
3.3.1. void add(T data)	15
3.3.2. public Aux<T>remove(String str)	17
3.3.3. public T search(String str) throws EntityNotFoundException	20
3.3.4. public boolean remove(T data)	21
3.3.5. Juntándolo todo en el catálogo: public Film<?>removeFilm(String str)	22
3.3.6. Otros casos	23
3.4. Casos de prueba y testeo unitario con JUnit5	24
3.4.1. Resultado	27
3.5. Comentarios	28
4. Fase 3: Hashing y grafos implícitos	29
4.1. Cambios con respecto a la fase 2	29
4.1.1. Changelog	29
4.1.2. Wrappers	30
4.2. Estructura general	31
4.2.1. Diagrama UML	32
4.3. Análisis de algoritmos y métodos.	32
4.3.1. public HashSet<Artist>getAdjacents()	32
4.3.2. public int getGraphDistance(String str1, String str2)	33
4.3.3. LinkedList<Artist>computeShortestPath(String str1, String str2)	35
4.4. Casos de prueba y testeo unitario con JUnit5	36
4.4.1. Resultado	37
4.5. Enunciado - apartado e)	38
4.6. Comentarios	38
5. Bibliografía	39

1. Introducción

Para esta primera fase del desarrollo de la aplicación de IMDB [1], se nos ha propuesto aplicar los conceptos presentados en clase en cuanto a lo que al análisis e implementación de algoritmos se refiere.

Lo interesante de este proyecto es que, es un primer y tímido acercamiento 'al mundo real' donde comen-zamos a ver la necesidad de trabajar en la optimización y eficiencia.

En este caso, tenemos que trabajar con grandes cantidades de datos proporcionado en formato fichero, cuya gestión, si nos basamos únicamente en un tratamiento de datos lineal y/o '*rudimentario*', hará que generaremos cuellos de botella o grandes latencias.

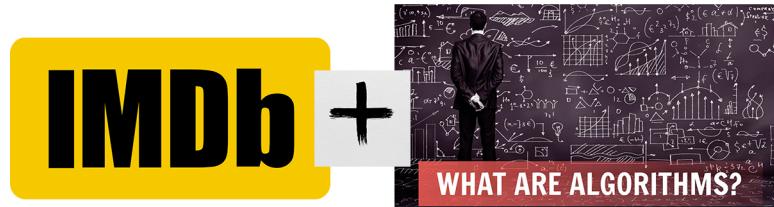


Figura 1: Intro

Es aquí donde observamos lo fundamental que resulta el elevar a un nivel superior el análisis previo al desarrollo, así como el ser minuciosos con las diferentes técnicas de búsqueda, ordenación etc.

2. Fase 1: Estructuras lineales y búsqueda dicotómica

2.1. Estructura general.

Para el desarrollo de ésta primera fase hemos diseñado una estructura general con varias capas de abstracción que permitan una fácil adaptación y con un grado de escalabilidad decente de cara a las próximas fases, diseñando así un proyecto extensible pero sencillo.

Esto nos permite trabajar con modelos y otros aspectos que aportan genericidad al sistema; siempre intentando seguir los principios de la **Ley de Demeter** en la medida de lo posible [8].

Veamos una imagen del árbol del directorio del proyecto y una breve explicación.

2.1 Estructura general. 2 FASE 1: ESTRUCTURAS LINEALES Y BÚSQUEDA DICOTÓMICA

```
[ basajaun@macbook ~] ~/Workspace/Projects/IMDB-app [ master ± ] tree -I "out"
.
├── IMDB-app.iml
├── LICENSE
└── README.md
├── data
│   ├── files
│   │   ├── cast.txt
│   │   └── films.txt
│   └── smallerfiles
│       ├── README
│       ├── cast_medium.txt
│       ├── cast_small.txt
│       ├── cast_tiny.txt
│       ├── films_medium.txt
│       ├── films_small.txt
│       └── films_tiny.txt
└── doc
    └── fase 1
        ├── UML
        │   └── IMDB-app.png
        └── base documents
            ├── CreacionProyecto.pdf
            ├── PracticaEDA.pdf
            └── PracticaEDAFase1.pdf
└── src
    └── main
        └── java
            ├── appIMDB.java
            ├── entities
            │   ├── Artist.java
            │   ├── Film.java
            │   └── models
            │       ├── DataModel.java
            │       └── Entity.java
            ├── exceptions
            │   ├── EmptyDataException.java
            │   ├── EntityNotFoundException.java
            │   ├── LoadMgrException.java
            │   └── NonValidInputValue.java
            ├── libs
            │   └── Stopwatch.java
            ├── managers
            │   ├── CatalogIMDB.java
            │   └── LoadMgr.java
            └── templates
                ├── DataWrapper.java
                └── SearchEngine.java
16 directories, 30 files
```

Figura 2: Estructura base

2.1 Estructura general. 2 FASE 1: ESTRUCTURAS LINEALES Y BÚSQUEDA DICOTÓMICA

■ **data**: ficheros-recursos que nutren nuestra fuente principal de información.

■ **doc**: documentos necesarios y/o requeridos para esta primera fase.

■ **src/main/java**: Contiene las siguientes clases/paquetes:

appIMDB.java: Aplicación principal encargada de inicializar y cargar la información, así como de la gestión del menú principal.

managers: Paquete que contiene las clases encargadas de la gestión del Catálogo y la carga y la vinculación de los datos.

libs Paquete donde se alojan las librerías (modificadas) de terceros.

templates: Paquete donde se implementan las clases genéricas que gestionarán estructuras y datos, sin realmente conocer su funcionamiento interno.

exceptions: Paquete donde se encuentran las excepciones definidas por nosotros mismos.

entities: Paquete que aloja las definiciones abstractas que toda entidad ha de poseer, y también las entidades que representan los modelos principales del proyecto (Películas e Intérpretes).

2.1.1. Sobre la ley de Demeter

Aprovechando que en la asignatura se ha hecho una pequeña introducción a los tipos de datos abstractos o TADs [3], hemos querido trabajar en ello intentando seguir el principio que dicta la **ley de Demeter** [8]. Ésta ley es una muy buena práctica de programación que repercute en un mejor mantenimiento del código, todo ello gracias a que reduce el nivel de dependencia que una clase tiene con las estructuras o funcionamientos internos de objetos de otras clases. Ésta práctica que tiene como objetivo el **acoplamiento entre clases** se puede llevar a cabo de muchísimas formas distintas, y los TADs nos pueden ayudar.

Alguno de los miembros del grupo tenía un poco de experiencia con templates de C++ [9], lo cual ha sido algo positivo pero a la vez un inconveniente tras el salto a los **generics** de Java, dadas las costumbres adquiridas con el tiempo.

No obstante, nos gustaría hacer especial mención por su enorme ayuda en éste y otros temas, al famoso libro **Clean Code** [Robert C. Martin] [4], así como también al libro de **Algorithms** [Robert Sedgewick | Kevin Wayne] [7].

Ley de Demeter: Un objeto no debería de conocer las entrañas de otros objetos con los que interactúa.

2.1 Estructura general. 2 FASE 1: ESTRUCTURAS LINEALES Y BÚSQUEDA DICOTÓMICA

2.1.2. Diagrama UML

Veamos el diagrama UML resultante (la imagen puede verse en la misma entrega del proyecto y/o en [Github](#) en mayor resolución):

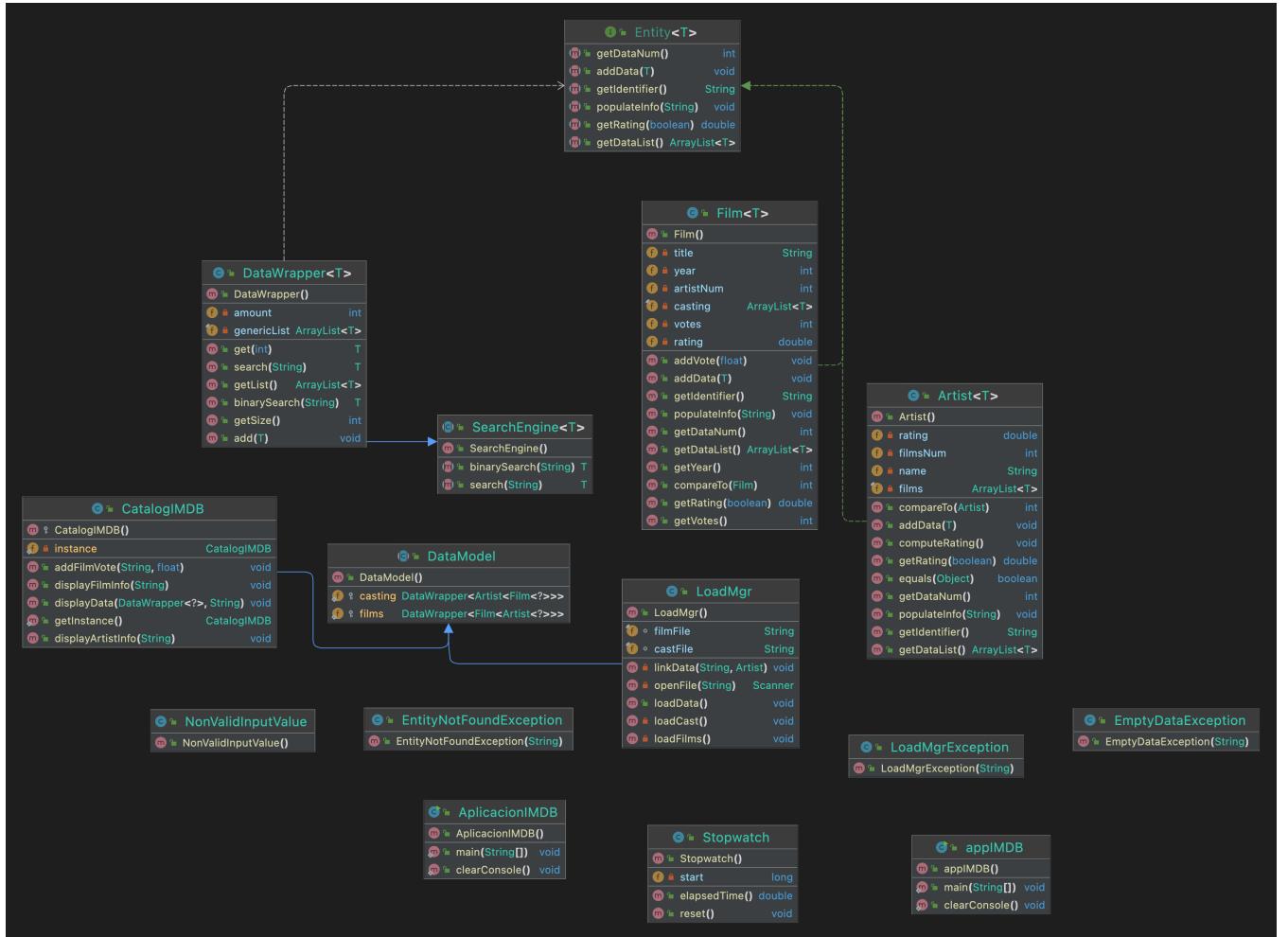


Figura 3: UML

2.2 Análisis de algoritmos y ESTRUCTURAS LINEALES Y BÚSQUEDA DICOTÓMICA

2.2. Análisis de algoritmos y métodos.

Hemos creado diferentes tipos de métodos que cubran las necesidades de cada una de las partes del código. Sin embargo vamos a mencionar los que consideramos algorítmicamente relevantes, ya que consideramos oportuno enfatizar los elementos que nos han proporcionado una mayor eficiencia con respecto lo estudiado previamente. Los tiempos que se mostrarán, son los obtenidos con la carga de los ficheros más grandes.

El código ha sido documentado siguiendo las pautas básicas de **JavaDoc** - planeamos extenderlo durante futuras fases.

2.2.1. Búsqueda

Para la búsqueda, el algoritmo base que utilizamos es el **Binary Search** o **Búsqueda Binaria**. Sin embargo hemos diseñado de manera adicional un algoritmo de búsqueda lineal, para hacer comparaciones.

No obstante, como ya explicaremos un poco más adelante, ahora mismo vamos a centrarnos únicamente en el propio método - de manera aislada, sin embargo depende del contexto y de cómo sea invocado para determinar su coste final.

Veamos primero un ejemplo de sus implementaciones:

```
/**
 * Búsqueda lineal a través de la lista genérica.
 * @param key Elemento a buscar.
 * @return El elemento encontrado.
 * @throws EntityNotFoundException lanzamos excepción en caso de no haber encontrado el elemento.
 */
public T search(String key) throws EntityNotFoundException {
    for(T item : genericList)
        if(item.getIdentifier().equalsIgnoreCase(key))
            return item;
    throw new EntityNotFoundException("[EXCEPTION] [SEARCH ENGINE] Entity not found: " + key);
}

/**
 * Búsqueda binaria o dicotómica sobre la lista genérica.
 * @param key Elemento a buscar.
 * @return Elemento encontrado.
 * @throws EntityNotFoundException lanzamos excepción en caso de no haber encontrado el elemento.
 */
public T binarySearch(String key) throws EntityNotFoundException {

    int first = 0;
    int last = amount-1;
    int middle = (first + last)/2;

    while(first <= last){
        if (genericList.get(middle).getIdentifier().compareTo(key) < 0)
            first = middle + 1;
        else if (genericList.get(middle).getIdentifier().equals(key))
            return genericList.get(middle);
        else
            last = middle - 1;

        middle = (first + last)/2;
    }
    throw new EntityNotFoundException("[EXCEPTION] [SEARCH ENGINE] Entity not found: " + key);
}
```

Figura 4: Search y Binary Search (Iterativo)

DataWrapper.java (extends `SearchEngine<T>`):

- `public T search(String key) throws EntityNotFoundException`

Algoritmo: Búsqueda lineal, recorre uno a uno los elementos hasta encontrar el coincidente.

Orden: $\mathcal{O}(n)$

- `public T binarySearch(String key) throws EntityNotFoundException`

Algoritmo: En cada iteración, se establece un elemento como primero y otro como último, de manera que se comprueba si el elemento que está posicionado en el medio de estos es el que buscamos. Si es mayor, entonces se buscará en la parte izquierda del array, o por el contrario, si el elemento del medio es menor, se buscará en la parte derecha. Se iterará hasta encontrar el elemento o hasta que se ha recorrido el array completamente (en cuyo caso, implica que **no existe**).

Orden: $\mathcal{O}(\log n))$

Como se ha mencionado previamente, el coste de realizar una búsqueda que invoque el método `binarySearch` vendrá determinado por su contexto. Por ejemplo, si el usuario decide realizar una búsqueda de una película o artista, el sistema simplemente realizará un **búsqueda binaria**, con lo que el orden siempre será de $\mathcal{O}(\log n))$, ya que, como veremos en el próximo apartado, hemos decidido sacrificar unos poco segundos tras la carga y vinculación de todos los datos para ordenar la colección de datos de los artistas.

En el siguiente apartado, veremos también otro de los posibles contexto donde éste método de búsqueda es invocado, cuyo orden o '**Big O**' será distinto.

2.2.2. Carga de datos

El proceso de carga de datos desde los ficheros de `casts.txt` y `films.txt` es el siguiente:

- Carga las películas de manera secuencial.
- Carga los artistas y relaciona con las películas ya cargadas (buscando mediante **búsqueda binaria** en la colección de películas) para cada uno de los nuevos intérpretes/artistas.

2.2 Análisis de algoritmos y ESTRUCTURAS LINEALES Y BÚSQUEDA DICOTÓMICA

Hemos desarrollado dos opciones, **dejando como base la primera**:

- **Opción 1:** Ordenar el `arraylist` de casting para poder hacer `búsqueda binaria` posteriormente. De ésta manera agregamos alrededor de 3.3-3.5s extra en la carga inicial (ficheros grandes) pero reducimos drásticamente el tiempo de búsqueda en cada una de las mismas. Mejorando así la experiencia de usuario (búsquedas de Artistas muy rápidas).
- **Opción 2:** NO ordenar el `arraylist` de casting y en la búsqueda utilizar el método `search` del `SearchEngine`. Eliminamos el tiempo extra de carga que necesita el ordenado, pero se incrementa el tiempo por cada búsqueda.

Ambas son válidas y funcionales.

LoadMgr.java:

- `private void loadFilms() throws IOException, LoadMgrException`

Algoritmo: Carga secuencial, cada película leída del fichero se agrega a un `ArrayList`.

Orden: $\mathcal{O}(n)$

Tiempo de ejecución: Media de 1.05s

- `private void loadCast() throws IOException, LoadMgrException`

Algoritmo: Carga secuencial, pero esta vez **cada vez que se carga un artista, se realiza una búsqueda** en la colección de películas mediante el algoritmo de `búsqueda binaria` para **cada una de las películas** en la que dich@ artista haya participado.

Orden: $\mathcal{O}(n(m \log p))$ [$n = \text{artistas}$, $m = \text{películas}/\text{artista}$, $p = \text{películas}$]

Tiempo de ejecución: Media de 47.016s

Veamos un pequeño fragmento de código correspondiente al método `linkData`, el cual es el encargado de crear la relación entre artistas y películas. Éste método es invocado por cada una de las películas en las que un determinado artista haya participado:

```
private void linkData(String filmName, Artist artist){  
    try{  
        Film currFilm = films.binarySearch(filmName);  
        artist.addData(currFilm);  
        currFilm.addData(artist);  
    } catch (EntityNotFoundException ignore){}  
}
```

2.3. Ordenación

Nos habíamos planteado la posibilidad de no realizar un ordenado de la colección de artistas, pero como se ha mencionado en el apartado anterior, hemos preferido sacrificar esos $3.3\text{-}3.5\text{s}$ extra en la carga y ordenarlo, con tal de mejorar la experiencia de usuario y poder proporcionar búsquedas realmente veloces (según [Stopwatch \[5\]](#), las búsquedas son de 0.0s), tanto para artistas como para películas.

Depende de lo que se necesite - hemos dejado ambas opciones implementadas y se pueden cambiar a demanda por el programador, eliminando el coste extra que genera la ordenación si se desea, pero incrementando a $0.3\text{-}0.5\text{s}$ las búsquedas de artistas (ya que se realizan mediante búsqueda lineal).

Dicho esto y en este caso, en lugar de implementar nosotros un algoritmo de ordenación y tras investigar sobre las distintas posibilidades que ofrece Java; nos hemos decantado por su **opción más estable**, que no es otra que la de utilizar el método `sort` del Framework `Collections`, el cual implementa el algoritmo de ordenado **Mergesort** (iterativo). Otras opciones eran atractivas, como la de `Array.sort()` que se nutre del veloz **Dual pivot Quicksort** [6], pero únicamente ordena de manera nativa datos primitivos y al parecer en según qué situaciones se considera inestable ya que pese a que su orden sea $\mathcal{O}(n \log n)$ puede cambiar a $\mathcal{O}(n \log n^2)$

2.3.1. Análisis (Mergesort iterativo)

Collections.java/List.java:

- `default void sort(Comparator<? super E>c)`

Algoritmo: Mergesort (iterativo) en cada iteración, el array se divide en 2 hasta llegar a un único elemento - el cual se considerará ordenado. De esta manera, se van uniendo todas las mitades en las que se han ido separando el array, comparando sus elementos, reduciendo drásticamente el número de comparaciones con respecto a otro tipo de algoritmos de orden de crecimiento superior (Bubble sort, insertion sort...etc).

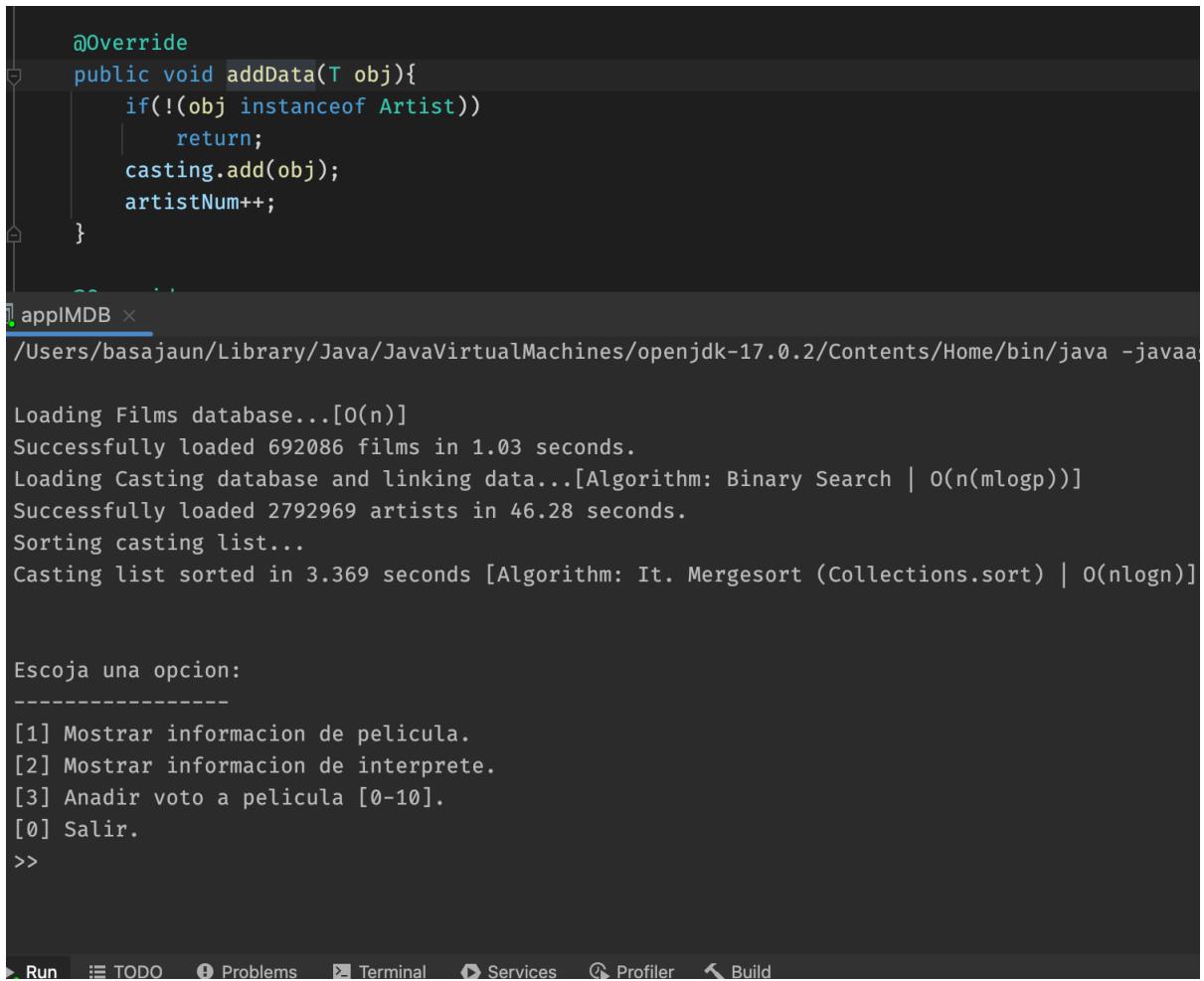
Orden: $\mathcal{O}(n \log n)$

Tiempo de ejecución: Media de 3.4s

2.4. Simulación (tiempos)

Llegados a este punto, veamos una simulación de la aplicación para poder observar un ejemplo de todo lo visto anteriormente en el análisis de algoritmos, y contrastar los tiempos.

Estos tiempos pueden diferir con una distancia moderada entre ejecuciones y, por supuesto, entre máquinas.



The screenshot shows a Java IDE interface with a code editor and a terminal window. The code editor displays a portion of a Java class with an overridden `addData` method. The terminal window shows the execution of a Java application named `appIMDB`. The application loads a database of films and artists, sorts the casting list, and presents a menu for user interaction. The menu options are:

- [1] Mostrar informacion de pelicula.
- [2] Mostrar informacion de interprete.
- [3] Anadir voto a pelicula [0-10].
- [0] Salir.

The terminal also shows the path to the Java executable and the command used to run the application.

Figura 5: Simulación app (tiempos)

2.5. Casos de prueba

Hemos realizado un exhaustivo testeo, el cual ha repercutido en una mejora del **tratamiento de excepciones** en la aplicación. En lugar de redactar directamente la cantidad ingente de pruebas y tests que hemos realizado, creemos que el código y la documentación muestran que hemos trabajado no solamente la congruencia de datos si no también posibles fallos de segmentación / crashes.

Nos gustaría haber podido implementar distintos test unitarios con **JUnit5**, pero no hemos dispuesto de tiempo suficiente (queda pendiente para futuras fases).

No obstante, para desarrollar el mencionado testeo, hemos realizado todo tipo de pruebas en cada una de las opciones. Por mencionar unas pocas:

- Introducir distintos símbolos y tipos de datos en cada una de las opciones (intentando producir excepciones que no estuvieran siendo capturadas).
- Valores fuera de rango en cada una de los posibles input.
- Comprobar una miríada de veces la congruencia de datos cargados (contrastando la información que nos mostraba la aplicación con la reflejada en los ficheros fuente).
- Agregar múltiples votos a una película en sesiones 'largas' y comprobar una vez más que los datos fueran congruentes; tanto en los posibles ratings de los artistas como de las películas, y ver cómo afectaban dichos cambios de manera cruzada entre artistas/películas.
- Etc.

2.6. Observaciones y comentarios

La algoritmia es un mundo tan complejo como apasionante. Hemos podido experimentar lo gratificante que resulta el proceso de desarrollo, al ver cómo mejorando la implementación del código y aplicar ciertos algoritmos, la eficiencia del sistema mejora exponencialmente.

Estamos deseando continuar - prepararemos durante los próximos días una nueva branch en nuestro Github para la siguiente fase.

3. Fase 2: Árboles binarios de búsqueda

Durante la fase 1 utilizamos ciertas estructuras de datos sobre las cuales aplicamos algoritmos de búsqueda como el de búsqueda binaria o dicotómica, de manera que conseguíamos optimizar mucho el proceso.

Sin embargo, como ya explicamos en el apartado [2.3](#), el listado de artistas / intérpretes no estaba ordenado, aunque nosotros decidimos realizar una ordenación y sacrificar unos pocos segundos en la carga de información.

Para solventar esto, se nos ha encargado el modificar la infraestructura del proyecto para poder incluir **árboles de búsqueda binarios**. Ya que, para la gestión de datos no ordenados, las búsquedas y demás operaciones que se aplican sobre éste tipo de estructuras son realmente eficientes gracias a la naturaleza de las mismas.

Siguiendo la línea de la fase 1 y con objeto de crear estructuras de datos genéricas independientes del tipo de dato que se esté tratando, hemos generalizado las mismas así como las clases que las poseen.

3.1. Cambios con respecto a la fase 1

3.1.1. Changelog

Se han hecho algunos cambios basándose en el feedback recibido tras la entrega de la fase 1. Como pequeño **Changelog**:

- Pasar la especificación de los nombres de los ficheros a la aplicación en lugar de en el mismo `LoadMgr.java`.
- Hemos mejorado el display de los artistas y películas. Ahora se mostrarán todos los artistas pese a que no posean rating. [Issue en Github](#).
- Ahora la aplicación permite introducir `floats` como votos.
- Hemos incrementado el número de decimales a mostrar para el rating (de 2 a 4).

3.1.2. Wrappers

Para poder adaptar el concepto de DataWrapper a las nuevas especificaciones, lo hemos refactorizado como **LinealWrapper**; que se encargará de una lista y operaciones sobre la misma. A su vez, para atender a los requerimientos de la nueva estructura de la lista de artistas, hemos creado un segundo wrapper: **BTreeWrapper** (*Binary Tree Wrapper*).

De ésta manera, las películas mantendrán una estructura lineal con búsquedas eficientes (binarias) ya que están ordenadas, y el casting, al no estarlo - será organizado mediante el wrapper de árbol binario, BTTreeWrapper.

La adaptación de lo desarrollado durante la fase 1 a ésta nueva fase ha sido muy sencilla y amena dado el diseño base. Los sistemas e infraestructuras creados nos han permitido 'ignorar' sus gestiones internas de manera que la migración haya sido posible cambiando unas pocas líneas de código, y además, en una capa muy superior. Así que, mediante una simple refactorización e implementando las nuevas estructuras en árbol e interfaz que explicaremos en el siguiente punto, hemos integrado a los sistemas ya existentes una nueva estructura. [1] [2]

Siguiendo la sugerencia de la creación de una nueva interfaz, hemos diseñado DataCollection, que tendrá el papel de especificar el comportamiento que una colección de datos ha de tener.

Básicamente es una interfaz que nos permite definir ciertas operaciones que toda colección de datos necesita implementar, independientemente de sus *intrigulis* e infraestructuras. Por ejemplo, en el caso de ésta segunda fase, los *wrappers* relacionados a las películas y artistas, pese a que uno posea una **estructura lineal** y el otro una **estructura de árbol**, pertenecerán a la interfaz DataCollection y así lo especificamos en su modelo de datos, que nos permitirá instanciar con el wrapper que posea la estructura que deseemos:

```

25
26     public interface DataCollection<T> {
27         void add(T node);
28         T search(String str) throws EntityNotFoundException;
29         T remove(String str) throws EntityNotFoundException;
30         boolean remove(T data) throws EntityNotFoundException;
31         int size();
32     }
33
34     /**
35      * Junto con Entity, esta clase conformara el modelo de datos
36      * que gestionara el Catálogo.
37     */
38
39     public abstract class DataModel {
40
41         protected static DataCollection<Film<Artist<?>>> films;
42         protected static DataCollection<Artist<?>> casting;
43
44         public DataModel(){}
45     }
46
47     protected CatalogIMDB(){
48
49         films = new LinealWrapper<>();
50         casting = new BTTreeWrapper<>();
51     }
52
53 }
```

Figura 6: DataCollection interface.

3.2. Estructura general.

Con esta segunda fase, la estructura de nuestro proyecto ha cambiado. Obviamos ciertos ficheros y la carpeta data que contiene los archivos con la información de las películas/artistas.

```
basajaun@macbook ~ /Workspace/Projects/IMDB-app phase-2 tree -I "out|data|*.iml|*.md|LICENSE"
.
+-- doc
    +-- fase 1
        +-- UML
            +-- IMDB-app.png
        +-- base documents
            +-- CreacionProyecto.pdf
            +-- PracticaEDA.pdf
            +-- PracticaEDAFase1.pdf
        +-- memoria
            +-- Memoria fase 1.pdf
    +-- fase 2
        +-- Base Documents
            +-- PracticaEdaFase2.pdf
        +-- UML
            +-- IMDB-app.png
    +-- images
        +-- IMDB-app.png
        +-- Intro.png
        +-- search.png
        +-- simulation.png
        +-- structure.png
+-- src
    +-- main
        +-- java
            +-- appIMDB.java
            +-- entities
                +-- Artist.java
                +-- Film.java
                +-- models
                    +-- DataCollection.java
                    +-- DataModel.java
                    +-- Entity.java
            +-- exceptions
                +-- EmptyDataException.java
                +-- EntityNotFoundException.java
                +-- LoadMgrException.java
                +-- NonValidInputValue.java
            +-- libs
                +-- Stopwatch.java
            +-- managers
                +-- CatalogIMDB.java
                +-- LoadMgr.java
            +-- templates
                +-- LinealWrapper.java
                +-- SearchEngine.java
            +-- scalable
                +-- BTreeWrapper.java
                +-- Node.java
    +-- test
        +-- java
            +-- managers
                +-- CatalogIMDBTest.java
            +-- templates
                +-- LinealWrapperTest.java
23 directories, 31 files
```

Figura 7: Estructura base (Phase 2)

3.2.1. Diagrama UML

Como consecuencia de los cambios, disponemos de un nuevo diarama UML resultante:



Figura 8: Diagrama UML (Phase 2)

3.3. Análisis de algoritmos y métodos.

Para dar soporte a ésta segunda fase se han implementado diversas clases e interfaces y con ello, los distintos métodos que describen sus acciones. Vamos a analizar aquellos métodos y algoritmos que inciden directamente sobre el propósito de la fase.

3.3.1. void add(T data)

Equivalente al método propuesto `public void eliminarInterprete(Interprete inter)`. Éste método ha sido implementado en las clases `BTreeWrapper` y `Node`, para poder agregar al árbol binario de artistas/intérpretes un nuevo intérprete.

BTreeWrapper.java

```

46     /**
47      * DataCollection overrides.
48      */
49     @Override
50     public void add(T node){
51         if(!isEmpty())
52             root.add(node);
53         else
54             root = new Node<T>(node);
55         numNodes++;
56     }

```

Figura 9: Método add en la clase BTreeWrapper.java

El algoritmo de naturaleza recursiva para agregar un nuevo nodo, es el siguiente:

Node.java

```

40
41     public void add(T data){
42         if(info.compareTo(data) < 0)
43             if(hasRight())
44                 right.add(data);
45             else
46                 right = new Node<T>(data);
47             else
48                 if(hasLeft())
49                     left.add(data);
50                 else
51                     left = new Node<T>(data);
52     }
53

```

Figura 10: Método add en la clase Node.java

Orden del método add:

- **Orden:** $\mathcal{O}(h)$ que realmente será $\mathcal{O}(\log n)$, y el peor de los casos $\mathcal{O}(n)$ [$h = \text{altura}$, $n = \text{artistas}$]

3.3.2. public Aux<T>remove(String str)

Equivalente al método propuesto `public Interprete eliminarInterprete(String nombre)`. Para eliminar un artista del árbol (y que aún persista en las listas de artistas de cada una de las películas), se ha desarrollado los siguientes métodos, el cual está basado directamente en los expuestos en las transparencias de la asignatura, ya que son óptimos y funcionales.

`BTreeWrapper.java`

```

65  /**
66   * Elimina un intérprete del árbol (puede seguir estando en las listas de intérpretes de las películas)
67   * @param str Nombre del intérprete a eliminar
68   * @return el Interprete (si se ha eliminado), null en caso contrario
69   */
70  @Override
71  public T remove(String str) throws EntityNotFoundException {
72      if(!isEmpty()){
73          T entity = root.remove(str).getInfo();
74          if(entity != null){
75              numNodes--;
76              return entity;
77          }
78          else
79              throw new EntityNotFoundException("[EXCEPTION] Entidad no encontrada");
80      }
81      throw new EntityNotFoundException("[EXCEPTION] Árbol vacío.");
82  }
83

```

Figura 11: Remove en `BTreeWrapper.java`

En el caso de la clase `Node.java` que veremos a continuación, hay que tener en cuenta que existe un proceso extra que tan sólo se ejecutará una vez; en el caso de que se haya encontrado el elemento y su nodo posea tanto hijo izquierdo como hijo derecho. En éste caso, realizaremos una **búsqueda extra del menor elemento del subárbol derecho**, cuyo orden será de: $\mathcal{O}(\log n)$ y el peor de los casos $\mathcal{O}(n)$. Veamos la implementación del método:

Node.java

```

81  /**
82   * Elimina un intérprete del árbol (puede seguir estando en las listas de intérpretes de las películas)
83   * Implementación de transparencias.
84   * @param str Nombre del intérprete a eliminar
85   * @return el Interprete (si se ha eliminado), null en caso contrario
86   */
87  @
88  public Aux<T> remove(String str){
89      int comp = str.compareTo(info.toString());
90      Aux<T> res = new Aux<>();
91
92      if(comp == 0){
93          res.info = info;
94          if(!hasLeft()){
95              res.node = right;
96              return res;
97          }
98          else if(!hasRight()){
99              res.node = left;
100             return res;
101         }
102         else{
103             Aux<T> min = right.removeMin();
104             right = min.node;
105             info = min.info;
106             res.node = this;
107             return res;
108         }
109     } else if(comp < 0){
110         if(hasLeft()){
111             res = left.remove(str);
112             left = res.node;
113         }
114         res.node = this;
115         return res;
116     } else{
117         if(hasRight()){
118             res = right.remove(str);
119             right = res.node;
120         }
121         res.node = this;
122     }
123 }
124

```

Figura 12: Implementación de remove basada en las transparencias.

Orden: $\mathcal{O}(h)$ que realmente será $\mathcal{O}(\log n)$, y el peor de los casos $\mathcal{O}(n)$ [$h = \text{altura}$, $n = \text{artistas}$]

Node.java

```

124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
    /**
     * Función auxiliar que elimina y devuelve el mínimo de del subárbol derecho.
     * @return Objeto de la clase auxiliar Aux.
     */
    public Aux<T> removeMin(){
        Aux<T> res = new Aux<>();
        if(!this.hasLeft()) {
            res.info = this.info;
            res.node = this.right;
        }else {
            Aux<T> resulLeft = this.left.removeMin();
            this.left = resulLeft.node;
            res.info = resulLeft.info;
            res.node = this;
        }
        return res;
    }

    class Aux<T> extends Comparable<T>> {
        private T info;
        private Node<T> node;
        public Aux(){}
        public T getInfo(){ return info; }
    }
}

```

Figura 13: Método `removeMin()` y su clase auxiliar.

Orden: $\mathcal{O}(h)$ que realmente será $\mathcal{O}(\log n)$, y el peor de los casos $\mathcal{O}(n)$ [$h = \text{altura}$, $n = \text{artistas}$]

Con lo que:

Orden del método `remove`:

- **Orden:** $\mathcal{O}(\log n) + \mathcal{O}(\log n) \approx \mathcal{O}(\log n)$

3.3.3. public T search(String str) throws EntityNotFoundException

Equivalente al método propuesto `public Interprete buscarInterprete(String nombre)`. Realiza una búsqueda sobre el árbol y devuelve un objeto de tipo T.

`BTreeWrapper.java`

```

57
58
59  @Override
60  public T search(String str) throws EntityNotFoundException {
61      if(!isEmpty())
62          return root.search(str);
63      return null;
64
65 }
```

Figura 14: Search en el wrapper

`Node.java`

```

54  public T search(String str) throws EntityNotFoundException {
55      if(info.toString().equals(str))
56          return info;
57
58      if(isLeaf())
59          throw new EntityNotFoundException("[EXCEPTION] Entidad no encontrada");
60
61      if(info.toString().compareTo(str) < 0)
62          if(hasRight())
63              return right.search(str);
64          else
65              throw new EntityNotFoundException("[EXCEPTION] Entidad no encontrada");
66
67      if(hasLeft())
68          return left.search(str);
69
70      throw new EntityNotFoundException("[EXCEPTION] Entidad no encontrada");
71 }
```

Figura 15: Algoritmo de búsqueda.

Orden del método `search`:

- **Orden:** $\mathcal{O}(h)$ que realmente será $\mathcal{O}(\log n)$, y el peor de los casos $\mathcal{O}(n)$ [$h = \text{altura}$, $n = \text{artistas}$]

3.3.4. public boolean remove(T data)

Equivalente al método propuesto `public boolean eliminarPelicula(Pelicula pel)`. Para éste método, y ya que hemos utilizado la estructura lineal basada en ArrayLists, haremos uso de una función propia de ésta colección. **Para ello necesitamos el objeto en sí** (en este caso de tipo `T`, pero como veremos en el siguiente punto, el dato que se nos proporciona para identificar la película en sí es el nombre, es decir, un **string**). Para ello, implementamos el paso intermedio haciendo así una **sobrecarga del método remove**: `public T remove(String str) throws EntityNotFoundException`.

`LinealWrapper.java`

```

52
53 ①@Override
54 ②    public T remove(String str) throws EntityNotFoundException {
55      T data = search(str);
56      if(remove(data))
57          return data;
58      throw new EntityNotFoundException("Ha ocurrido un error, no se ha podido borrar.");
59
60 ③@Override
61 ④    public boolean remove(T data){ return genericList.remove(data); }

```

Figura 16: Remove lineal en la clase `LinealWrapper`

- **search(String str)**: El objetivo de este `search` (que utiliza la búsqueda binaria o dicotómica que implementamos en la primera fase, apartado 2.2.1) es el de obtener el elemento en sí, para poder utilizar el método `remove(obj)` de la colección `ArrayList`. **Orden**: $\mathcal{O}(\log n)$.
- **remove(T data)** Una vez obtenido el elemento, podemos utilizar el método `remove(obj)` de `ArrayLists` y proceder al borrado. Éste método se encargará de su reorganización. Los `ArrayLists` realizan una búsqueda **lineal** para determinar el elemento a borrar, con **Orden**: $\mathcal{O}(n)$.

Existe un pequeño truco que no hemos realizado para así poder 'hacer uso' del `binary search` implementado durante la fase 1, pero eliminaría cierta redundancia. Consistiría en la creación de un objeto '*fake*' con el mismo nombre que el pasado como parámetro, aprovechándonos de que, internamente, la colección `ArrayLists` hará uso de `equals`. Con lo que, si tenemos implementado dicho método se contrastará que el objeto pasado como parámetro y el que está siendo analizado del `ArrayList` sean iguales (en este caso, en base al `string`).

Esto podría迫使 tal que : `.remove(new Clase(str))`. Igualmente, por los motivos indicados y por el hecho de estar utilizando tipos genéricos (no podemos instanciar los mismos), hemos decidido primero buscar el elemento y luego borrarlo mediante el método de la colección `ArrayLists`. No obstante, existen maneras de poder realizarlo.

Orden del método `remove` (lineal):

- **Orden**: $\mathcal{O}(\log n) + \mathcal{O}(n) \approx \mathcal{O}(n)$

3.3.5. Juntándolo todo en el catálogo: public Film<?>removeFilm(String str)

Llegados a este punto, podemos analizar el método encargado de procesar la eliminación de una película. Equivalente al método propuesto `public Pelicula eliminarPelicula(String titulo)`. Aquí se incluye también un método que sería el equivalente al `public boolean eliminarPelicula(Pelicula pel)` (primera línea de código), aunque devolvemos la película en sí, para después poder acceder a los distintos wrappers.

CatalogIMDB.java

```

157     /**
158      * Método encargado de borrar una película tanto de la lista/wrapper de películas
159      * como de las listas/wrappers propios de cada uno de los artistas que han participado
160      * en la misma. Se fuerza una actualización del rating en dichos artistas si aún existen
161      * películas que hayan participado. Se eliminan los artistas en caso contrario.
162      * @param str Nombre de la película a eliminar.
163      * @return devuelve referencia a objeto con la película eliminada.
164      * @throws EntityNotFoundException en caso de que no exista la película/artista buscado.
165      * @throws EmptyDataException Si se intenta realizar la operación sobre una estructura de artistas vacía.
166      */
167     public Film<?> removeFilm(String str) throws EntityNotFoundException, EmptyDataException {
168         Film<?> deletedFilm = films.remove(str);
169         try{
170             for(Entity<?> artist : deletedFilm.getDataList()){
171                 LinealWrapper<?> wrapper = artist.getWrapper();
172                 wrapper.remove(str);
173                 if(wrapper.isEmpty())
174                     casting.remove(artist.getIdentifier());
175                 else
176                     ((Artist<?>)artist).computeRating();
177             }
178             System.out.println("[EXCEPTION] Se ha borrado la película: " + deletedFilm.getIdentifier());
179         } catch(EmptyDataException e){
180             System.out.println("[EXCEPTION] No tiene artistas");
181         }
182         return deletedFilm;
183     }

```

Figura 17: Gestión del borrado de una película.

Si analizamos y consideramos el **peor** de los casos en cada situación, para cada uno de los métodos descritos anteriormente:

- **remove** - Una sola vez, para n películas: $\mathcal{O}(\log n) + \mathcal{O}(n) \approx \mathcal{O}(n)$ (por *orden de crecimiento*).
- Para los m artistas de dicha película: $\mathcal{O}(m)$
 - Se buscan la película en las p películas de cada artista y se borra: $\mathcal{O}(\log p) + \mathcal{O}(p) \approx \mathcal{O}(p)$
 - Si la lista de películas del artista está vacía se borra el artista del árbol - **remove**: $\mathcal{O}(\log r)$
 - Si no está vacía se recalcula el rating, recorriendo para ello todas las p películas de cada artista (teniendo en cuenta que se ha eliminado una): $\mathcal{O}(p - 1) \approx \mathcal{O}(p)$

Como sabemos que hay muchos más artistas que películas, para la rama condicional consideraremos como **peor** de los casos el caso en el que todos los artistas de la película queden con 0 películas tras el borrado. Lo que implica que **se deberá de realizar el borrado en el árbol m veces**.

Quedando el orden tal que:

Orden del método remove en catálogo:

- **Orden:** $\mathcal{O}(n) + \mathcal{O}(m * p * \log r)$ Aunque, si consideramos el peor de los casos, será $\mathcal{O}(n) + \mathcal{O}(m * p * r)$ [$n =$ total películas, $m =$ artistas de la película borrada, $p =$ películas de dichos artistas, $r =$ total artistas]

3.3.6. Otros casos

Existen otros métodos que se han requerido implementar. En la interfaz `DataCollection` se ha especificado el método `public int size()`, el cual, en el caso del `LinearWrapper` devolverá el tamaño del `ArrayList` mediante `list.size()` y en el caso del `BTtreeWrapper`, se devolverá el **número de nodos** del árbol, que se ha ido incrementando/decrementando conforme se añaden o eliminan los mismos.

El orden de éste método sin importar el wrapper que lo esté implementando (`LinearWrapper.java` ó `BTtreeWrapper.java`), será de **orden constante** $\mathcal{O}(1)$, al igual que el siguiente.

Otro es el caso del método `public void setInterpretes(InterfaceInterpretes interpretes)`, el cual recibe como parámetro un objeto que implemente la interfaz `DataCollection`, siendo el tipo de dato con el que los métodos de la interfaz trabajarán, el tipo perteneciente a la clase `Artista`, la cual tratará un wrapper cuyo tipo - en ésta parte del código - se desconoce:

`CatalogIMDB.java`

```

40      ↗    public static void setCasting(DataCollection<Artist<?>> newCast){
41          ↗        casting = newCast;
42      ↘

```

Figura 18: `setCasting` en `CatalogIMDB`

3.4 Casos de prueba y testeo unitario con JUnit5 FASE 2: ÁRBOLES BINARIOS DE BÚSQUEDA

3.4. Casos de prueba y testeo unitario con JUnit5

Al igual que en la primera fase, hemos sometido la aplicación a numerosos casos de prueba, favoreciendo el tratamiento de excepciones y creando así una aplicación robusta y carente de errores. La mayoría los hemos hecho conforme se iba desarrollando la aplicación y de manera manual.

Entre los distintos casos de prueba, hemos utilizado el expuesto en el enunciado de la fase 2 y, de manera adicional hemos escrito diferentes *pruebas unitarias* utilizando JUnit5 [11], para su automatización.

Los tests unitarios pueden encontrarse en la siguiente ruta del proyecto:

```
src/test/managers/CatalogIMDBTest.java
```

Antes de comenzar los tests, se ejecutará el método estático `static void setUp()` (contiene la cláusula `@BeforeAll` para ello), que se encargará de cargar los ficheros (utilizar los **grandes**) e inicializar ciertas variables.

```
CatalogIMDBTest.java
```

```
24      @BeforeAll
25      static void setUp() {
26          cat = CatalogIMDB.getInstance();
27          try{
28              LoadMgr loadMgr = new LoadMgr("files/films", "files/cast");
29              loadMgr.loadData();
30          } catch(LoadMgrException e){
31              System.out.println(e.getMessage());
32          }
33          initFilmSize = cat.getFilms().size();
34          initCastSize = cat.get_casting().size();
35      }
```

Figura 19: Tests setup.

Existe también otro método que se ejecutará al finalizar los tests (cláusula `@AfterAll`) y nos mostrará el tamaño de nuestras colecciones de datos, que serán distintos de los obtenidos durante la carga inicial.

Hemos mantenido el diseño de los tests muy simple, y únicamente hemos considerado si tienen que lanzar o no excepciones, o incluso si los valores devueltos son los esperados. Se contemplan casos como votar películas, eliminar películas y/o artistas, contrastar congruencias entre los datos esperados y los obtenidos.

```
addFilmVoteTest()
```

```
37      @Test
38      public void addFilmVoteTest(){
39          Assertions.assertThrows(EntityNotFoundException.class, () -> cat.addFilmVote("Non existent film", 8));
40          Assertions.assertThrows(NonValidInputValue.class, () -> cat.addFilmVote("Fight Club", 12));
41          |
42          Assertions.assertDoesNotThrow(()-> cat.addFilmVote("Fight Club", 9));
43          Assertions.assertDoesNotThrow(()-> cat.addFilmVote("I Love Sydney", 4));
44      }
```

Figura 20: Probar a votar una película

3.4 Casos de prueba y testeo unitario con JUnit FASE 2: ÁRBOLES BINARIOS DE BÚSQUEDA

```
removeFilmTest()
```

```
46  @Test
47  void removeFilmTest() throws EmptyDataException, EntityNotFoundException {
48
49      /*
50       * Intento de eliminación de películas no existentes, deben lanzar excepción.
51       */
52      Assertions.assertThatThrownBy(() -> cat.removeFilm("Wrong film"))
53          .isInstanceOf(EntityNotFoundException.class);
54
55      /*
56       * Me aseguro de la existencia de un artista en concreto.
57       */
58      Assertions.assertThatThrownBy(() -> cat.getCastings().search("Chiesatypo, Ricardo"))
59          .isInstanceOf(EntityNotFoundException.class);
60      Assertions.assertThat(() -> cat.getCastings().search("Vera, Brandon"))
61          .doesNotThrow();
62
63      /*
64       * Eliminación de película existente, no debe de lanzar excepción.
65       */
66      Assertions.assertThat(() -> cat.removeFilm("Fights"))
67          .doesNotThrow();
68
69      /*
70       * NOTA: Necesita los ficheros grandes.
71       * El artista Rotstein, Sebastian ha participado en 2 películas.
72       */
73      Assertions.assertThat(cat.getCastings().search("Rotstein, Sebastian").getWrappers().size())
74          .isEqualTo(3);
75
76      /*
77       * Nos aseguramos de la existencia de una película.
78       */
79      Assertions.assertThat(() -> cat.getFilms().search("Filmatron"))
80          .doesNotThrow();
81
82      /*
83       * Obtenemos el número de películas de una actriz que ha participado en
84       * Filmatron y que sabemos en una película más (ergo, 2).
85       */
86      int artistNumFilms = cat.getCastings().search("Setton, Carolina").getWrappers().size();
87
88      /*
89       * La borramos.
90       */
91      Film<?> film = cat.removeFilm("Filmatron");
92      modFilmSize++;
93      modCastSize += 2; //2 artistas van a ser eliminados al ser Filmatron su única película
```

Figura 21: Distintas pruebas al borrar películas

3.4 Casos de prueba y testeo unitario con JUnit FASE 2: ÁRBOLES BINARIOS DE BÚSQUEDA

```

92     /*
93      * Buscamos la película de manera explícita, se ha de lanzar una excepción.
94      */
95     Assertions.assertThrows(EntityNotFoundException.class, () -> cat.getFilms().search(film.getIdentifer()));

96
97     /*
98      * Después de las 2 eliminaciones, debemos tener 998 películas (films_tiny.txt).
99      * Si se utilizan los ficheros grandes: 692084 películas y 2792967 artistas.
100     */
101    Assertions.assertEquals(initFilmSize - modFilmSize, cat.getFilms().size());
102    Assertions.assertEquals(initCastSize - modCastSize, cat.get_casting().size());

103
104    /*
105     * Los artistas Chiesa, Rocardo y Goncalves, Luciano - al únicamente haber participado en la película anterior
106     * son eliminados.
107     */
108    Assertions.assertThrows(EntityNotFoundException.class, () -> cat.get_casting().search("Chiesa, Ricardo"));
109    Assertions.assertThrows(EntityNotFoundException.class, () -> cat.get_casting().search("Goncalves, Luciano-"));

110
111    /*
112     * Comprobamos que la actriz cuyo número de películas hemos almacenado antes y eran 2,
113     * Sigue existiendo:
114     */
115    Assertions.assertDoesNotThrow(() -> cat.get_casting().search("Setton, Carolina"));

116
117    /*
118     * Y ahora, que efectivamente, consta que únicamente ha participado en 1 película tras el
119     * borrado de Filmatron.
120     */
121    Assertions.assertEquals(artistNumFilms - 1, cat.get_casting().search("Setton, Carolina").getWrapper().size());

122
123    /*
124     * Despues del borrado de Filmatron, en nuestro sistema ha de constar que Rotstein, Sebastian
125     * ha participado en 2 películas.
126     */
127    Assertions.assertEquals(2, cat.get_casting().search("Rotstein, Sebastian").getDataList().size());
128 }

```

Figura 22: Distintas pruebas al borrar películas.

addFilmTest() y artistSearchTest()

```

133     @Test
134     void addFilmTest(){
135         Film film = new Film();
136         film.populateInfo("Random new Film\t2022\t-1\t-1");
137         cat.getFilms().add(film);
138         modFilmSize--;
139         Assertions.assertEquals(initFilmSize - modFilmSize, cat.getFilms().size());
140     }

141
142     @Test
143     void artistSearchTest(){
144         Assertions.assertThrows(EntityNotFoundException.class, () -> cat.get_casting().search("Non existant artist"));
145         Assertions.assertDoesNotThrow(() -> cat.get_casting().search("Morris, Nato"));
146     }
147

```

Figura 23: Testeo de adición de película y búsqueda de un artista.

3.4 Casos de prueba y testeo unitario con JUnit5 FASE 2: ÁRBOLES BINARIOS DE BÚSQUEDA

```
testAddAndLinkArtist()
```

The screenshot shows a code editor with a dark theme. The code is written in Java and contains a single test method:

```
155     @Test
156     void addAndLinkArtistTest() {
157         Assertions.assertDoesNotThrow(() -> {
158             Artist artist = new Artist();
159             artist.populateInfo("Artist, New");
160
161             artist.addData(cat.getFilms().search("Fight Science"));
162             cat.get_casting().add(artist);
163             modCastSize--;
164             Assertions.assertEquals(initCastSize - modCastSize, cat.get_casting().size());
165
166             Artist aQuery = cat.get_casting().search("Artist, New");
167             Film film = (Film)aQuery.getWrapper().get(0);
168             film.addData(aQuery);
169
170             Assertions.assertTrue(aQuery.getWrapper().search(film.getIdentifer()) != null
171                     && film.getDataList().contains(aQuery));
172         });
173     }
```

Figura 24: Agregar un artista y vincular a una película (mutuamente)

3.4.1. Resultado

Como podemos observar, si lanzamos los tests - pasan todos, lo cual implica que nuestros casos de prueba han sido correctos y los sistemas subyacentes han operado correctamente.

```
5/5 tests passed
```

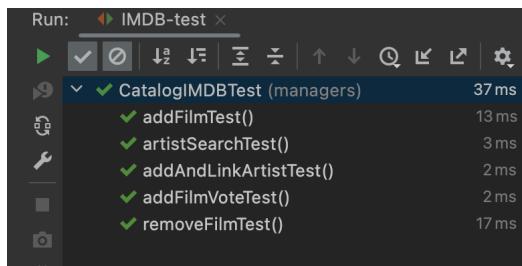


Figura 25: Todos los test han pasado las pruebas correctamente.

3.5. Comentarios

No hemos dispuesto de mucho tiempo, pero nos habría gustado utilizar un patrón de diseño basado en callbacks para la implementación de esta segunda fase: el patrón de diseño Observer [10]. Ya que, en casos como el que ocurre cuando se borra una película, si tuviésemos un observador, podría notificar a los objetos de los respectivos artistas que han participado en la misma de que ésta ha sido borrada - y así, gestionarse en consecuencia. Esto lo podríamos haber extrapolado a otras situaciones, claro está.

Igualmente, teniendo en cuenta que el proyecto ha adoptado una forma un poco distinta de la propuesta, hemos considerado no introducir nuevas capas de abstracción o complicar la infraestructura. Pero, ¡queda pendiente!

4. Fase 3: Hashing y grafos implícitos

De árboles binarios, a estructuras basadas en tablas de **hashing** y grafos. En éste caso, grafos *implícitos* ya que tratamos ciertas estructuras como si fueran grafos, pese a no estar implementados mediante la estructura tradicional de vértice - arista.

En ésta fase hemos visto cómo se puede incrementar el rendimiento y eficiencia del sistema de manera colosal, reduciendo las complejidades u ordenes temporales de los algoritmos a utilizar de órdenes como $\mathcal{O}(\log n)$ a órdenes constantes $\mathcal{O}(1)$, gracias a la estructura de las **tablas hash**.

También hemos utilizado algoritmos de búsqueda como el Breadth-first search (BSF), extremadamente útiles para encontrar elementos en estructuras de elementos relacionales como grafos, así como el menor del camino entre los mismos.

4.1. Cambios con respecto a la fase 2

4.1.1. Changelog

Estos son los cambios que hemos realizado, algunos basándonos en el feedback recibido tras la entrega de la fase 2. Como pequeño **Changelog**:

- Se nos pedía utilizar el método `setCasting()` (`setInterpretes`) de la clase `CatálogoIMDB.java` en la clase de la aplicación. Hemos preparado un ejemplo de cómo sería su uso, se muestra más abajo.
- Creación de una nueva clase llamada `HashMapWrapper.java`, la cual contiene un hashmap encargado de almacenar los intérpretes, y los métodos a implementar de la interfaz `DataCollection`.
- Para poder adaptar la nueva estructura con `HashMaps`, dadas la estructura y arquitectura que se ha ido desarrollando a lo largo del proyecto a través de los wrappers, clases abstractas e interfaces, ha sido muy sencillo. En `CatalogIMDB.java`, simplemente instanciar la variable casting como el nuevo wrapper: `casting = new HashMapWrapper<>();`

```
public static void main(String[] args){
    try{
        CatalogIMDB cat = CatalogIMDB.getInstance();

        final String filmFile = "smallerfiles/films_small";
        final String castFile = "smallerfiles/cast_small";

        LoadMgr loadMgr = new LoadMgr(filmFile, castFile);
        loadMgr.loadData();

        DataCollection<Artist<?>> casting = new LinealWrapper<>();
        CatalogIMDB.setCasting(casting);

        casting = new BTeeWrapper<>();
        CatalogIMDB.setCasting(casting);

        casting = new HashMapWrapper<>();
        CatalogIMDB.setCasting(casting);
    }
}
```

Figura 26: Ejemplo de uso de las 3 posibilidades de `setCasting()` en `AppIMDB.java`

4.1.2. Wrappers

Ésta vez, únicamente hemos agregado un nueva clase como wrapper, llamada: `HashMapWrapper.java`, la cual tiene como miembro el `hashmap` cuya clave será producto de hashear un string (función hash no implementada por nosotros, lo hará Java con su método `java.lang.String.hashCode()`) y el valor será de cualquier tipo, definido como tipo genérico `T` que almacenará los intérpretes y hará uso de los métodos definidos en la interfaz `DataCollection`:

```

27 import java.util.HashMap;
28
29 public class HashMapWrapper<T> implements DataCollection<T> {
30
31     HashMap<String, T> hashMap;
32
33     public HashMapWrapper() { hashMap = new HashMap<>(); }
34
35     @Override
36     public void add(T data) { hashMap.put(data.toString(), data); }
37
38     @Override
39     public T search(String str) throws EntityNotFoundException {
40         if(hashMap.isEmpty())
41             throw new EntityNotFoundException("[EXCEPTION] Emtpy HashMap");
42
43         T obj = hashMap.get(str);
44         if(obj == null)
45             throw new EntityNotFoundException("[EXCEPTION] Entit not found.");
46         return obj;
47     }
48
49     @Override
50     public T remove(String str) throws EntityNotFoundException {
51         if(hashMap.isEmpty())
52             throw new EntityNotFoundException("[EXCEPTION] Emtpy HashMap");
53         return hashMap.remove(str);
54     }
55
56     @Override
57     public boolean remove(T data) throws EntityNotFoundException {
58         if(hashMap.isEmpty())
59             throw new EntityNotFoundException("[EXCEPTION] Emtpy HashMap");
60         return hashMap.remove(data.toString(), data);
61     }
62
63     @Override
64     public int size() { return hashMap.size(); }
65
66 }
67
68
69 }
```

Figura 27: Nueva clase `HashMapWrapper.java`

4.2. Estructura general.

Con esta tercera fase, la estructura de nuestro proyecto ha cambiado mínimamente. Obviamos ciertos ficheros, la carpeta data que contiene los archivos con la información de las películas/artistas y los directorios de las fases anteriores.

```
basajaun@macbook ~ ~/Workspace/Projects/IMDB-app phase-3 tree -I "out|data|*.iml|*.md|LICENSE|LaTeX|images|fase 1|fase 2"
.
├── doc
│   └── fase 3
│       ├── base documents
│       │   └── Práctica - fase 3.pdf
│       └── memoria
│           └── UML
│               └── IMDB-app-fase-3.png
└── src
    ├── main
    │   └── java
    │       ├── appIMDB.java
    │       ├── entities
    │       │   ├── Artist.java
    │       │   ├── Film.java
    │       │   └── models
    │       │       ├── DataCollection.java
    │       │       ├── DataModel.java
    │       │       ├── Entity.java
    │       │       └── Wrapper.java
    │       └── graphs
    │           └── Entity.java
    ├── exceptions
    │   ├── EmptyDataException.java
    │   ├── EntityNotFoundException.java
    │   ├── LoadMgrException.java
    │   └── NonValidInputValue.java
    ├── libs
    │   └── Stopwatch.java
    ├── managers
    │   ├── CatalogIMDB.java
    │   └── LoadMgr.java
    ├── templates
    │   ├── LinealWrapper.java
    │   ├── SearchEngine.java
    │   └── hashing
    │       └── HashMapWrapper.java
    └── scalable
        ├── BTreeWrapper.java
        └── Node.java
    └── test
        └── java
            └── managers
                └── CatalogIMDBTest.java
```

Figura 28: Estructura base (Phase 3)

4.2.1. Diagrama UML

Como consecuencia de los cambios, disponemos de un nuevo diarama UML resultante:

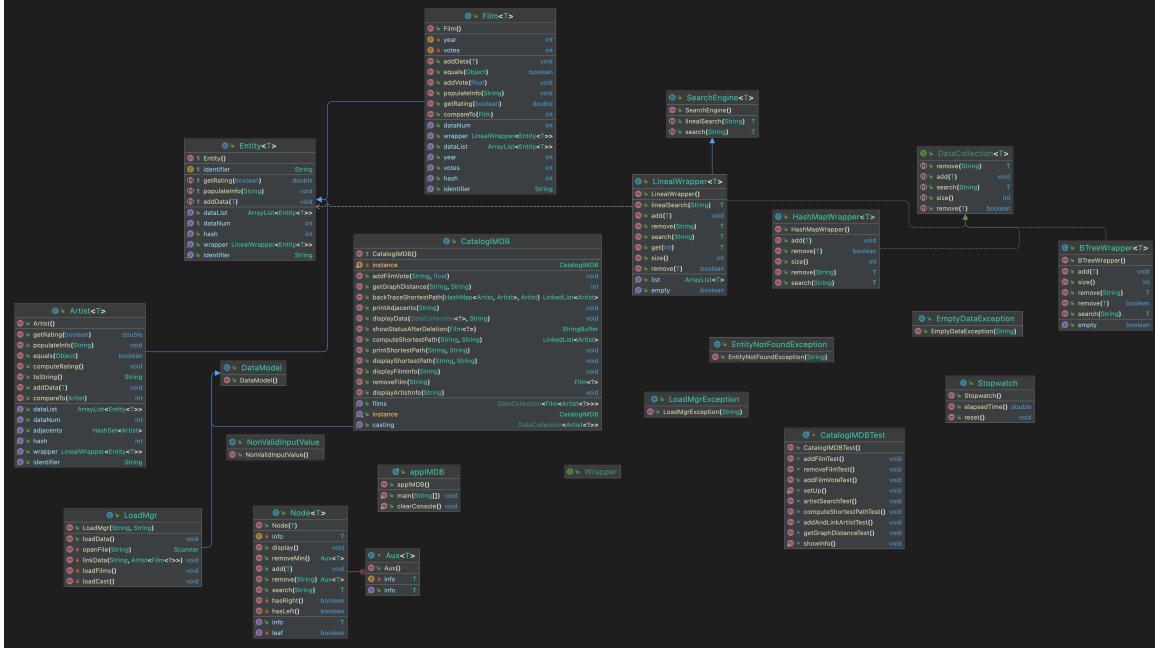


Figura 29: Diagrama UML (Phase 3)

4.3. Análisis de algoritmos y métodos.

Vamos a analizar aquellos métodos y algoritmos que inciden directamente sobre el propósito de la fase:

4.3.1. `public HashSet<Artist>getAdjacents()`

Primero veamos rápidamente el nuevo método en la clase `Artist.java` que nos permitirá obtener su *nodos* adjacentes. Para un determinado artista, se considerarán adjacentes todos los artistas que hayan participado en alguna de las películas el mismo. Es decir, un artista tendrá $m * n$ intérpretes adjacientes. Siendo m el número de películas el artista, y n el número de intérpretes de cada película.

`Artist.java`

```

34
35     public HashSet<Artist> getAdjacents(){
36         HashSet<Artist> hashSet = new HashSet<>();
37
38         /*
39          * No es necesario comprobar si existe o no cada actor que se quiera agregar.
40          * HashSet utiliza una función hash para comprobar que no esté duplicado, en
41          * el momento de agregar un nuevo elemento al conjunto.
42          */
43         for(Object film: linealWrapper.getList())
44             for(Object artist : ((Film<?>)film).getWrapper().getList())
45                 if(!((Artist)artist).getIdentifier().equals(identifier))
46                     hashSet.add((Artist)artist);
47
48         return hashSet;
49     }

```

Figura 30: Método `getAdjacents()` en la clase `Artist.java`

Orden del método `getAdjacents`:

- **Orden:** $\mathcal{O}(m * n)$ [$m = \text{películas}$, $n = \text{artistas}$]

4.3.2. `public int getGraphDistance(String str1, String str2)`

Método implementado en la clase `CatalogIMDB.java`, con objeto de obtener la distancia entre dos intérpretes. Se utiliza el algoritmo **Breadth-first search**.

Partiendo del intérprete base y para cada intérprete o vértice adjacente, se tendrá que analizar todos los adjacentes de los mismos hasta encontrar el intérprete que buscamos *llegar*. Para lo cual, en la especificación de su orden veremos el peor de los casos.

CatalogIMDB.java

```

198     /**
199      * Distancia entre 2 artistas.
200      * No estamos trabajando con un grafo en sí, pero vamos
201      * a simular su funcionamiento.
202      *
203      * Algoritmo: Bread-first search.
204      */
205     public int getGraphDistance(String str1, String str2) throws EntityNotFoundException {
206         HashSet<Artist> visited = new HashSet<>();
207         Queue<Artist> queue = new LinkedList<>();
208         HashMap<Artist, Integer> distancesMap = new HashMap<>();
209
210         Artist init = casting.search(str1);
211         Artist dest = casting.search(str2);
212         queue.add(init);
213         distancesMap.put(init, 0);
214
215         while(!queue.isEmpty()){
216             Artist currentArtist = queue.poll();
217
218             if(currentArtist.getIdentifier().equals(dest.getIdentifier()))
219                 return distancesMap.get(currentArtist);
220
221             HashSet<Artist> adjacents = currentArtist.getAdjacents();
222             for(Object adjArtist : adjacents){
223                 if(!visited.contains((Artist)adjArtist)){
224                     queue.add((Artist)adjArtist);
225                     visited.add((Artist)adjArtist);
226                     distancesMap.put((Artist)adjArtist, distancesMap.get(currentArtist) + 1);
227                 }
228             }
229         }
230         throw new EntityNotFoundException("Something went wrong.");
231     }

```

Figura 31: Método getGraphDistance() en la clase CatalogIMDB.java

Orden del método getGraphDistance:

- **Orden:** $\mathcal{O}(h+a+(h*p*n))$ que será el peor de los casos, donde cada uno de los intérpretes visitará todos sus adjacentes (tan sólo una vez, de ahí la suma) [$h = \text{artistas-en-tabla-hash}$, $a = \text{adjacentes}$, $m = \text{películas}$, $n = \text{artistas}$]

4.3.3. `LinkedList<Artist>computeShortestPath(String str1, String str2)`

Método implementado en la clase `CatalogIMDB.java`, con objeto de mostrar el *path* o camino más corto entre dos determinados artistas. También se utiliza el algoritmo **Breadth-first search**, para luego *backtrackear* o rastrear hacia atrás el camino que se ha generado, gracias a que vamos registrando qué nodo es anterior de cada nodo en el hashmap llamado `backtraceMap`, conforme vamos buscando el elemento que hará que el algoritmo pare.

`CatalogIMDB.java`

```

245     public LinkedList<Artist> computeShortestPath(String str1, String str2) throws EntityNotFoundException {
246         Queue<Artist> queue = new LinkedList<>();
247         HashMap<Artist, Artist> backTraceMap = new HashMap<>();
248
249         Artist init = casting.search(str1);
250         Artist dest = casting.search(str2);
251
252         queue.add(init);
253         backTraceMap.put(init, null);
254
255         while(!queue.isEmpty()){
256             Artist current = queue.poll();
257             if(current.equals(dest))
258                 return backTraceShortestPath(backTraceMap, current);
259
260             for(Object adj : current.getAdjacents()){
261                 if(!backTraceMap.containsKey((Artist)adj)){
262                     queue.add((Artist)adj);
263                     backTraceMap.put((Artist)adj, current);
264                 }
265             }
266         }
267         throw new EntityNotFoundException("[EXCEPTION] Entity not found");
268     }
269
270     public LinkedList<Artist> backTraceShortestPath(HashMap<Artist, Artist> backTraceMap, Artist init){
271         LinkedList<Artist> shortestPath = new LinkedList<>();
272
273         while(init != null){
274             shortestPath.add(init);
275             init = backTraceMap.get(init);
276         }
277         Collections.reverse(shortestPath);
278         return shortestPath;
279     }

```

Figura 32: Métodos `computeShortestPath()` y `backTraceShortestPath` en la clase `CatalogIMDB.java`

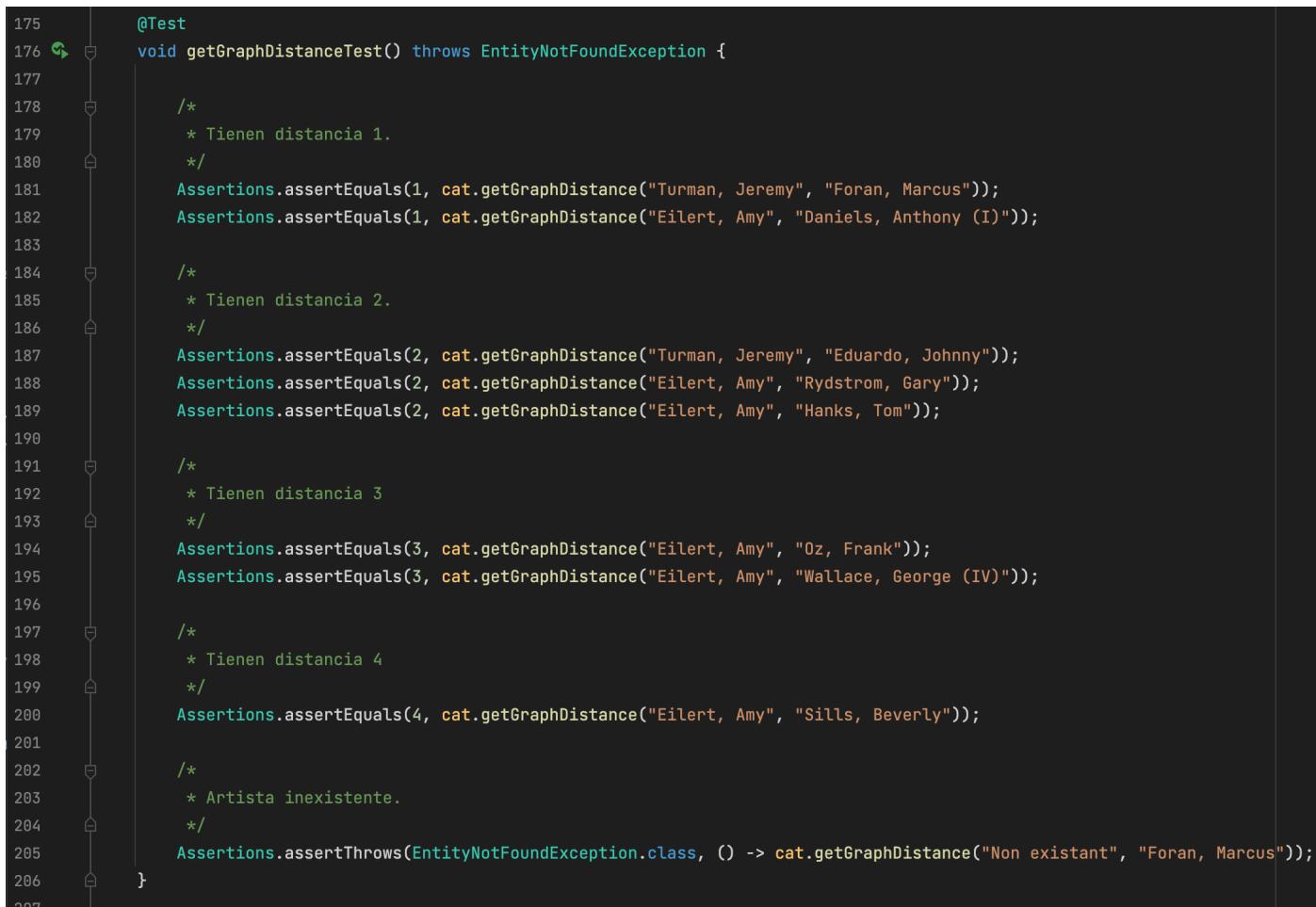
Orden del método `computeShortestPath`:

- **Orden backTraceShortestPath:** $\mathcal{O}(h)$ que será el peor de los casos, en el caso de que el camino más corto (por ende, el hashmap `backTraceMap` esté constituido por todos los interprétes o vértices de la estructura. $[h = \text{artistas} - \text{en} - \text{tabla} - \text{hash}]$
- **Orden:** $\mathcal{O}(2h + a + (h * p * n))$ que será el peor de los casos, el `h` extra es por las iteraciones que se harán para **reconstruir** el camino después de encontrar el elemento final buscado. No obstante, es equivalente a $\mathcal{O}(h + a + (h * p * n))$ por orden temporal. $[h = \text{artistas} - \text{en} - \text{tabla} - \text{hash}, a = \text{adjacentes}, m = \text{películas}, n = \text{artistas}]$

4.4. Casos de prueba y testeo unitario con JUnit5

En ésta tercera parte, como siempre - hemos ido depurando y corrigiendo errores conforme iban saliendo. No obstante, hemos sometido a la aplicación a numerosos tests de forma manual. De manera adicional y, como hicimos ya en la anterior fase - se han desarrollado unos métodos a modo de testing para automatizar pruebas.

```
getGraphDistanceTest()
```



```

175
176     @Test
177     void getGraphDistanceTest() throws EntityNotFoundException {
178
179         /*
180          * Tienen distancia 1.
181          */
182         Assertions.assertEquals(1, cat.getGraphDistance("Turman, Jeremy", "Foran, Marcus"));
183         Assertions.assertEquals(1, cat.getGraphDistance("Eilert, Amy", "Daniels, Anthony (I)"));
184
185         /*
186          * Tienen distancia 2.
187          */
188         Assertions.assertEquals(2, cat.getGraphDistance("Turman, Jeremy", "Eduardo, Johnny"));
189         Assertions.assertEquals(2, cat.getGraphDistance("Eilert, Amy", "Rydstrom, Gary"));
190         Assertions.assertEquals(2, cat.getGraphDistance("Eilert, Amy", "Hanks, Tom"));
191
192         /*
193          * Tienen distancia 3
194          */
195         Assertions.assertEquals(3, cat.getGraphDistance("Eilert, Amy", "Oz, Frank"));
196         Assertions.assertEquals(3, cat.getGraphDistance("Eilert, Amy", "Wallace, George (IV)"));
197
198         /*
199          * Tienen distancia 4
200          */
201         Assertions.assertEquals(4, cat.getGraphDistance("Eilert, Amy", "Sills, Beverly"));
202
203         /*
204          * Artista inexistente.
205          */
206         Assertions.assertThrows(EntityNotFoundException.class, () -> cat.getGraphDistance("Non existant", "Foran, Marcus"));
207     }

```

Figura 33: Probar distintas opciones para obtener la distancia.

4.4 Casos de prueba y testeo unitario con JUnit54 FASE 3: HASHING Y GRAFOS IMPLÍCITOS

```
removeFilmTest()
```

```

208     @Test
209     void computeShortestPathTest() throws EntityNotFoundException {
210
211         /*
212          * Shortest path:
213          * Demar, Diana - Porter, Christopher S. - Silvia, Carissa
214          */
215         StringBuilder path = new StringBuilder();
216         for(Artist a : cat.computeShortestPath("Demar, Diana", "Silvia, Carissa"))
217             path.append(a.getIdentifier());
218         Assertions.assertTrue(path.toString().contains("Porter, Christopher S."));
219
220         /*
221          * Shortest path:
222          * Eilert, Amy - Allers, Roger - Burtt, Ben - Oz, Frank
223          */
224         path = new StringBuilder();
225         for(Artist a : cat.computeShortestPath("Eilert, Amy", "Oz, Frank"))
226             path.append(a.getIdentifier());
227         Assertions.assertTrue(path.toString().contains("Allers, Roger"));
228         Assertions.assertTrue(path.toString().contains("Burtt, Ben"));
229
230         /*
231          * Shortest path:
232          * Eilert, Amy - Tiffany (I) - Minnelli, Liza - Welch, Raquel - Sills, Beverly
233          */
234         path = new StringBuilder();
235         for(Artist a : cat.computeShortestPath("Eilert, Amy", "Sills, Beverly"))
236             path.append(a.getIdentifier());
237         Assertions.assertTrue(path.toString().contains("Tiffany (I)"));
238         Assertions.assertTrue(path.toString().contains("Minnelli, Liza"));
239         Assertions.assertTrue(path.toString().contains("Welch, Raquel"));
240     }

```

Figura 34: Pruebas para la elección del camino más corto.

4.4.1. Resultado

Como podemos observar, si lanzamos los tests - pasan todos, lo cual implica que nuestros casos de prueba han sido correctos y los sistemas subyacentes han operado correctamente.

2/2 tests passed

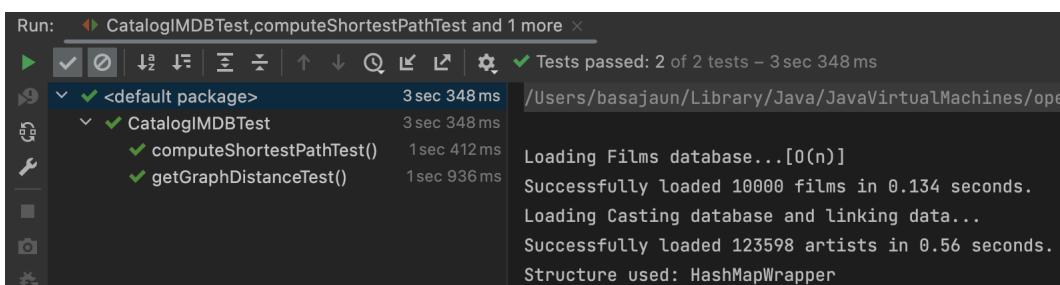


Figura 35: Ambos tests han pasado la prueba satisfactoriamente.

4.5. Enunciado - apartado e)

Creemos que sería muy positivo el utilizar una tabla **hash** para la **lista de películas** ya que, aunque gracias a que esté ordenada logramos el orden de $\mathcal{O}(\log n)$ mediante búsqueda binaria o dicotómica, si cambiásemos su estructura podríamos obtener búsquedas de orden constante $\mathcal{O}(1)$ en cada petición que se haga durante el proceso de carga de intérpretes.

4.6. Comentarios

Hemos disfrutado muchísimo del desarrollo de éste proyecto y, nos parece que su enunciado y planteamiento tienen una estructura muy bien pensada y apropiada para la asignatura.

El ir aplicando los conceptos vistos en clase de manera progresiva y el ir pudiendo ver de manera totalmente progresiva cómo mejora el rendimiento de un programa conforme vamos aprendiendo nuevas estructuras de datos y algoritmos, ha sido realmente enriquecedor.

El uso de tablas hash deja atrás todo lo visto con anterioridad. Claro está, depende del contexto y las necesidades que la arquitectura de una aplicación requiera. Pero podemos decir que es una estructura de datos que utilizaremos con asiduidad de ahora en adelante - se nos ha abierto un mundo. Así como con los grafos, vemos muchísimas posibilidades de aplicaciones reales y estamos deseando indagar más en estos temas.

5. Bibliografía

Referencias

- [1] https://es.wikipedia.org/wiki/Internet_Movie_Database
- [2] https://en.wikipedia.org/wiki/Law_of_Demeter
- [3] https://es.wikipedia.org/wiki/Tipo_de_dato_abstracto
- [4] https://en.wikipedia.org/wiki/Robert_C._Martin
- [5] <https://introcs.cs.princeton.edu/java/32class/Stopwatch.java.html>
- [6] https://www.researchgate.net/publication/283532116_Why_Is_Dual-Pivot_Quicksort_Fast
- [7] <https://algs4.cs.princeton.edu>
- [8] https://en.wikipedia.org/wiki/Law_of_Demeter
- [9] [https://en.wikipedia.org/wiki/Template_\(C%2B%2B\)](https://en.wikipedia.org/wiki/Template_(C%2B%2B))
- [10] https://en.wikipedia.org/wiki/Observer_pattern
- [11] https://en.wikipedia.org/wiki/Unit_testing

«*Programar es comprender.*»

–Kristen Nygaard

7 de Diciembre de 2022