

IMDB-app (fase 1)

Memoria del proyecto.

Aingeru García
Alba Gonzalez Zarpón
Sergio de los Toyos Sandoval

Índice

1. Introducción	2
2. Estructura general.	2
2.1. Sobre la ley de Demeter	4
2.2. Diagrama UML	5
3. Análisis de algoritmos y métodos.	6
3.1. Búsqueda	6
3.1.1. Análisis	7
3.2. Carga de datos	7
3.2.1. Análisis	8
3.3. Ordenación	9
3.3.1. Análisis (Mergesort iterativo)	9
3.4. Simulación (tiempos)	10
4. Casos de prueba	11
5. Observaciones y comentarios	11
6. Bibliografía	12

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 comenzamos 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 generemos cuellos de botella o grandes latencias.

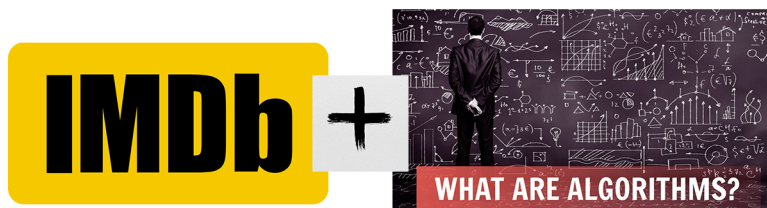


Figura 1: Intro

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. 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.

```
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
basajaun@macbook ~/Workspace/Projects/IMDB-app master ±
```

Figura 2: Estructura base

- **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. 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.

2.2. 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

3. 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.

3.1. 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)

3.1.1. Análisis

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 itera 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 pocos 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 contextos donde éste método de búsqueda es invocado, cuyo orden o '**Big O**' será distinto.

3.2. 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.

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.

3.2.1. Análisis

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 dicho artista haya participado.

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

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 fileName, Artist artist){
    try{
        Film currFilm = films.binarySearch(fileName);
        artist.addData(currFilm);
        currFilm.addData(artist);
    } catch (EntityNotFoundException ignore){}
}
```

3.3. 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 3.3-3.5s 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 0.3-0.5s 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 era 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 que situaciones se considera inestable ya que pese a que su orden sea $\mathcal{O}(n \log n)$ puede cambiar a $\mathcal{O}(n \log n^2)$.

3.3.1. 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 ésta 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).

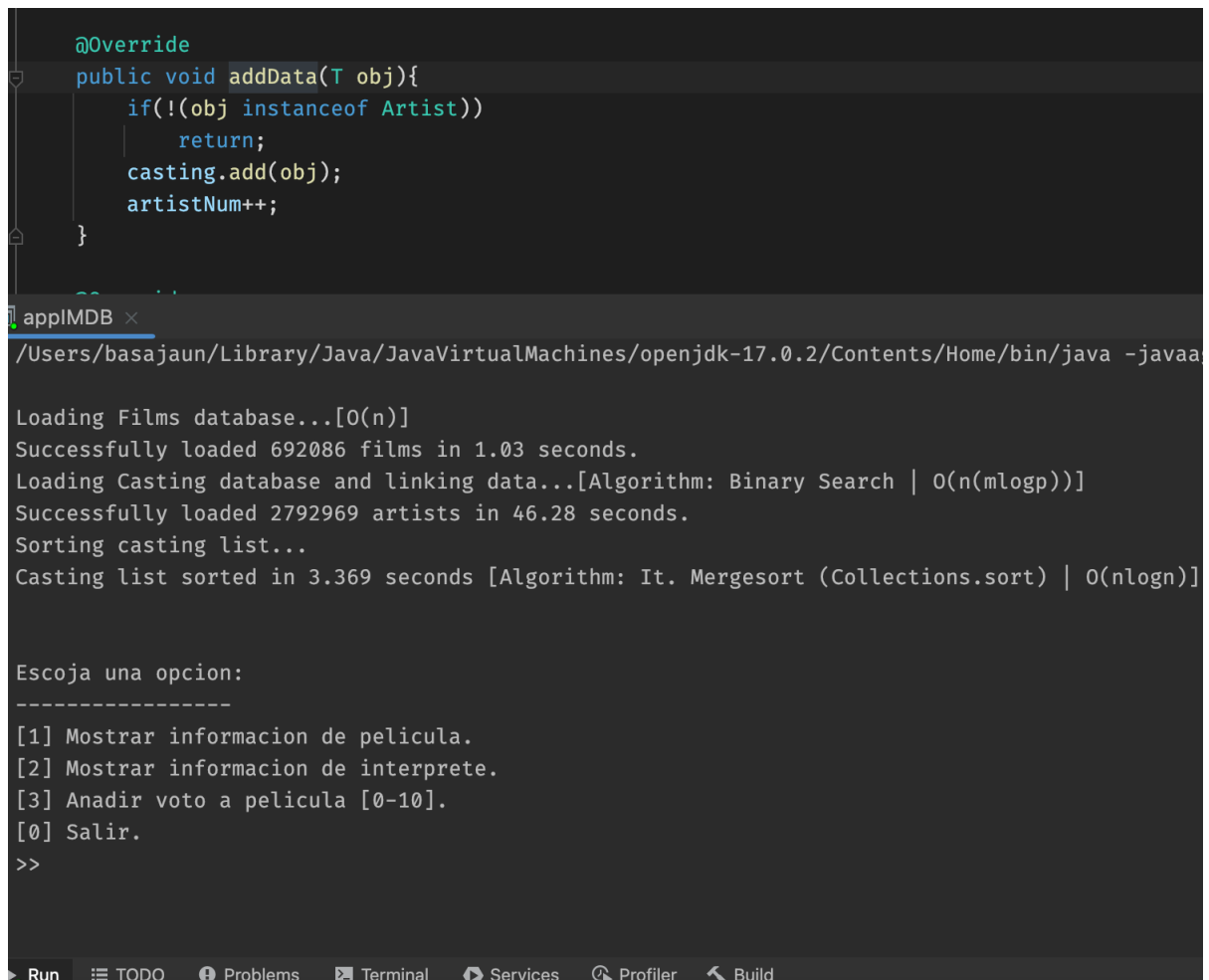
Orden: $\mathcal{O}(n \log n)$

Tiempo de ejecución: Media de 3.4s

3.4. 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 image shows a screenshot of an IDE. The top part displays a Java code snippet for a method `addData` that checks if an object is an instance of `Artist` and then adds it to a casting list. Below the code, a terminal window titled `appIMDB` shows the execution of the application. The terminal output includes the path to the Java executable, the loading of the films and casting databases with their respective execution times, the sorting of the casting list, and a menu of options for the user to interact with the application.

```
@Override
public void addData(T obj){
    if(!(obj instanceof Artist))
        return;
    casting.add(obj);
    artistNum++;
}

/Users/basajaun/Library/Java/JavaVirtualMachines/openjdk-17.0.2/Contents/Home/bin/java -javaa

Loading Films database...[O(n)]
Successfully loaded 692086 films in 1.03 seconds.
Loading Casting database and linking data...[Algorithm: Binary Search | O(n(mlogp))]
Successfully loaded 2792969 artists in 46.28 seconds.
Sorting casting list...
Casting list sorted in 3.369 seconds [Algorithm: It. Mergesort (Collections.sort) | O(nlogn)]

Escoja una opcion:
-----
[1] Mostrar informacion de pelicula.
[2] Mostrar informacion de interprete.
[3] Anadir voto a pelicula [0-10].
[0] Salir.
>>
```

Figura 5: Simulación app (tiempos)

4. 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.

5. 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.

6. 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))

«*Programar es comprender.*»

–Kristen Nygaard

25 de Octubre de 2022