Fecha

09-11-2015

Peio Valle  
Sergio Erlantz Tobal

GestIon imdb

De Actores y Peliculas

Contenidos

[Introducción y descripción general 3](#_Toc431554231)

[Diseño de las clases 3](#_Toc431554232)

[Estructuras de datos principales y alternativas examinadas 3](#_Toc431554233)

[Diseño e implementación de los algoritmos principales 4](#_Toc431554234)

[Método buscarElemento 4](#_Toc431554235)

[Código 4](#_Toc431554236)

[Conclusiones y resultados empíricos de las pruebas 5](#_Toc431554237)

# Introducción y descripción general

Se nos pide un programa en el cual recojamos de un fichero una cantidad enorme de nombres, tanto de actores/actrices como de películas para guardarlos en una estructura de datos y trabajar con ellos. En el archivo estarán juntos tanto los nombres como los títulos de las películas separados por ###. Se deberá poder buscar actores, películas, borrar, insertar… Basicamente todas aquellas funciones que se suelen ver en una base de datos y se llaman CRUD (Create, Read, Update, Delete).

# Estructuras de datos principales y alternativas examinadas

En un principio se pensó en usar usar estructuras de datos lo mas eficientes posibles, asi pues estuvimos mirando las 3 colecciones mas conocidas en Java, las listas, los sets y los maps. Veámoslos en una comparativa.

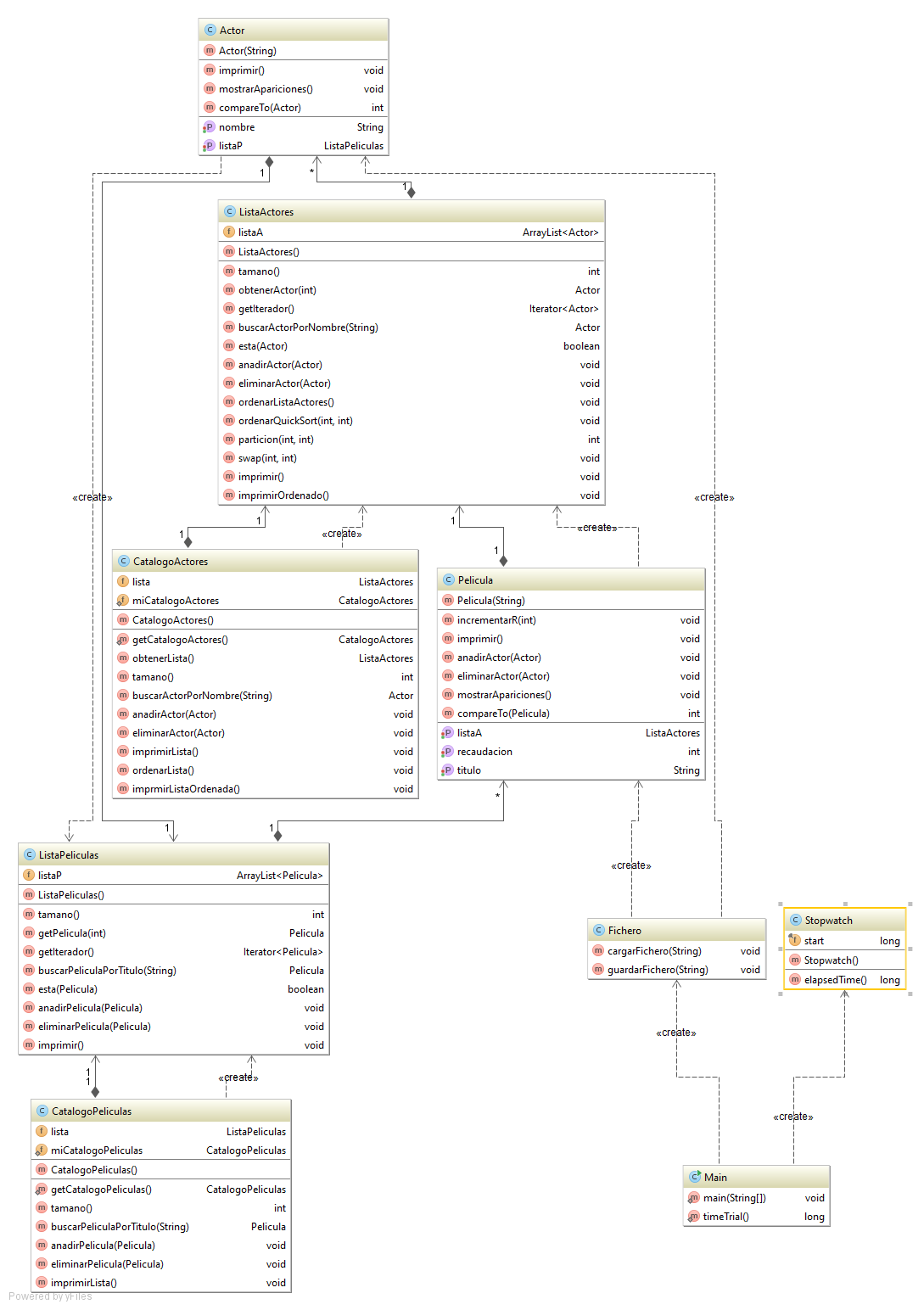
|  |  |  |  |
| --- | --- | --- | --- |
|  | Lists | Maps | Sets |
| Admiten Duplicidad | Si | Keys únicas, valores Si | No |
| Permitir elementos null | Si | Solo 1 key, valores todos los que se quiera | Solo 1 debido a la no duplicidad |
| Ordenadas | Si | No, porque va por keys | No ordenadas |
| Implementaciones | Arraylist, Linkedlist, Vector | Hashmap, LinkedHashmap, Hashtable, Treemap… | Hashset, LinkedHashset, TreeSet, SortedMap, SortedSet… |

Aun asi algunas implementaciones no cumplen lo mostrado en la tabla, por ejemplo Hashtable no admite keys o valores nulos o TreeMap y TreeSet que mantienen un orden impuesto por usando Comparator o Comparable.

Tambien lo hemos estado mirando en función del tiempo que tardarían las distintas operaciones en cada tipo de estructuras, y aunque queríamos implementar arboles ya que nunca habíamos trabajado con ellos al final se nos convenció que lo mejor era empezar con un Arraylist que la hemos usado anteriormente y sabemos su funcionamiento.

Por otro lado estuvimos mirando en algoritmos de ordenación cual seria mas eficiente y por lo tanto cual usar, el primero que nos planteamos era el Timsort, el cual se creo en 2002 con ideas de mergeSort (crear subarrays y ordenarlos cuando se juntan) y de insertionSort (cojer un elemento y lo mueve mirando los demás elementos y colocándolo ordenadamente en el array), y que su fundamental diferencia es que busca subarrays ordenados para que el tiempo sea menor, aunque si el array es demasiado pequeño se tiende a usar directamente InsertionSort. Desechamos este algoritmo ya que la forma de usarlo es llamar a clases ya hechas de Java y no sabíamos si esto estaba permitido y nos decantamos por el QuickSort que es el que vimos en clase que más rendimiento nos daba, aunque el mergesort es el mas estable.

# Diseño de las clases



Como se puede ver hemos creado clases para los modelos, las listas y los catalogos, asi como una clase para el cronometro y otra para la gestión del fichero. Los catalogos son clases Singleton que tienen listas de objetos que a su vez tienen los modelos. Por ultimo estarían las clases auxiliares de Fichero, que se encarga de las funciones de guardar y cargar archivos, y el Stopwatch que es el cronometro para el tiempo.

# Diseño e implementación de los algoritmos principales

En este apartado se presentarán los métodos principales, indicando por cada uno su especificación (precondiciones y postcondiciones), además de los casos de prueba planteados. Por cada método no trivial, se presentará la descripción del algoritmo implementado. Finalmente, se deberá presentar, de manera razonada, el cálculo del coste de cada algoritmo.

Presentamos un ejemplo de una posible manera de presentar un método de ejemplo.

## Método buscarElemento

public boolean buscarElemento(int x) {

/\* Precondición: el array no está vacío

/\* Postcondición: el resultado es true si x pertenece a la lista y false si no

Casos de prueba:

* Búsqueda de un elemento en lista vacía
* Búsqueda de un elemento en lista de un solo elemento:
  + el elemento buscado es igual al de la lista
  + el elemento buscado no es el de la lista
* Búsqueda en una lista no vacía
  + El elemento buscado no está en la lista
  + El elemento buscado es el primero de la lista
  + El elemento buscado es el último de la lista
  + El elemento buscado está en la mitad de la lista

Implementación del algoritmo:

i := 0

enc := 0

mientras (i <= n) y no enc hacer

si el elemento i-ésimo es igual a x

entonces enc := true

si no i := i + 1

devolver enc

Coste: el algoritmo, en el caso peor, examina todos los elementos de la lista. Para una lista de n elementos, el coste será O(n).

# Código

















En este apartado se presentará el código de las clases, junto con los programas de prueba o Junits desarrollados.

Se puede insertar un índice eligiendo “Insertar” → “Índices” → “Índices” y después la pestaña “Índice” (opción “índice de contenido”).

# Conclusiones y resultados empíricos de las pruebas

Aquí se presentarán las principales conclusiones obtenidas del trabajo: brevemente se explicarán los resultados principales y características principales de la solución desarrollada. También se podrán indicar las posibles líneas de trabajo futuras.

Además, puede servir para indicar las principales dificultades y problemas encontrados.