Práctica II

Algoritmos y estructuras de datos

María Del Carmen Cortés Navarro

# Análisis y descripción de la aplicación

**\*\*Se presentan en esta memoria todas las clases para la ejecución de la aplicación tanto desde consola como mediante una interfaz gráfica desarrollada con Swing y Awt.**

Las clases se han estructurado en 4 paquetes: InputOutput, Lógica, Interfaz y Excepciones. Esto se ha realizado para separar las clases según su funcionalidad general.

El paquete **Interfaz** contiene una sola clase:

* **Menu:** Es la clase principal del programa. Su método main controla la ejecución de la aplicación.

Clase que hereda de JFrame e implementa la interfaz ActionListener. Esto es así porque genera una interfaz gráfica para representar el menú principal de la aplicación. Las opciones del menú se seleccionan mediante botones que generan eventos que la propia clase escucha. El método actionPerformed(ActionEvent e) es el que se encarga de escuchar estos eventos y ejecutar diferentes acciones dependiendo de qué botón lo haya generado.

El resto de los métodos de la clase permiten mostrar ventanas emergentes para realizar las diferentes acciones del menú.

En el paquete **InputOutput** se encuentran 2 clases principales:

* **ArchivosCSV:** Cuya principal función es la gestión y manejo de ficheros. Las excepciones lanzadas por sus métodos se capturan y gestionan desde la clase principal. Para leer y guardar valores en los archivos se decidió utilizar flujos de datos de texto. Los valores se almacenarán en formato CSV.
* **Consola:** se encarga, principalmente, de la entrada de datos desde el teclado y de la interacción con el usuario. Por último, sus métodos lanzarán (y relanzarán) excepciones que se capturarán y manejarán desde la clase principal (Main).
* **ArchivosCsvGui:** esta clase tiene la misma función que ArchivosCSV, pero está adaptada para el uso de la interfaz gráfica. Los mensajes que genera informando sobre procesos ejecutados correctamente o errores los presenta mediante ventanas emergentes, no con mensajes por consola.

El paquete Excepciones contiene las clases relacionadas con la generación de excepciones:

* **ErrorType (enum):** que define diferentes tipos de errores que pueden ocurrir durante la ejecución del programa. Por ejemplo, que no haya ninguna población abierta, que no se encuentre cierto ratón después de un proceso de búsqueda, etc.
* **DoNotExistException**: se creó una excepción propia del programa que será lanzada por los distintos métodos. Esta se ha utilizado para aunar diferentes excepciones lanzadas por métodos de librerías externas en una única excepción común al programa. Además de para gestionar cuando, en los algoritmos de búsqueda, no se encuentren o no existan objetos de tipo Raton, PoblacionRatones o File.
* **ErrorFileAccessException:** esta excepción se creó para ser lanzada cuando ocurriese algún error o interrupción en el flujo de entrada o salida de datos en/desde los archivos.
* **IllegalArgument:** Esta excepción se ha utilizado para aunar diferentes excepciones relacionadas con la entrada de valores inválidos, principalmente al crear instancias de los objetos o al realizar pruebas de validación sobre los datos.
* **GUIExceptions:** Excepción que se creó para representar errores en la interacción con la interfaz gráfica.
* **GUIError:** Enumerador que representa posibles errores al interactuar con una interfaz gráfica

En el paquete **Lógica** se encuentran, además de la clase principal, las clases creadas para administrar las poblaciones de ratones y las familias:

* **Main**: (Sería la clase principal si se quisiese ejecutar el programa por consola) Gestiona el flujo principal del programa, controla que se ejecuten las acciones correctas según la opción del menú elegida por el usuario y captura y maneja las excepciones.

Contiene 2 listas públicas que almacenan las clases File con las localizaciones de los archivos abiertos y las poblaciones creadas. Ambas listas son paralelas. Se decidió utilizar listas en vez de arrays para poder inicializarlas sin un valor inicial y poder añadir tantos elementos como se quiera.

* **PoblacionRatones**: Se encarga de almacenar la información de cada población, además de almacenar los ratones de dicha población en una lista privada.

A su vez, permite, mediante métodos públicos, añadir nuevos ratones a la población, devolver un ratón en específico, eliminar ratones de la población y devolver un String con la información de la población, generar una cadena de texto de tipo CSV con los datos de la población, crear una población a partir de una cadena de texto en formato CSV, realizar la simulación de Montecarlo, etc.

Esta clase **implementa la interfaz Comparable<T>** para poder definir un criterio de comparación que permitirá ordenarlo en una estructura de tipo árbol.

Contiene 2 constructores:

* + Uno general para inicializar una población especificando el nombre de la población, del responsable, y el número de días que sus ratones estarán en la instalación.
  + Otro que permite crear una población virtual basándose en algunos porcentajes especificados por el usuario.
* **Raton**: almacena las propiedades de cada ratón. Contiene dos métodos constructores.
  + Uno que permite especificar si los cromosomas estarán mutados o no. Atendiendo al sexo y a este parámetro, el constructor generará dos objetos de tipo Cromosoma (enum).
  + El otro permite pasar por parámetro los cromosomas. El constructor se encargará de validar que estos corresponden con el sexo del ratón.

Particularidades:

* + Para almacenar los cromosomas se decidió usar un array de tipo Cromosomas con 2 posiciones. Este se inicializa al indicar el sexo del ratón. Si es hembra, se almacenarán dos cromosomas X, y si es macho, uno X y otro Y.
  + En el método toString se realizan comprobaciones del tipo de cromosoma (X, Xmut, Y, Ymut) para comprobar si los ratones tienen alguna patología.
* **Cromosoma (enum):** limita la elección de cromosomas a cuatro opciones: dos tipos de cromosomas (X, Y) con y sin mutación (Xmut, Ymut).
* **Sexo (enum):** se creó un enumerador de tipo Sexo para restringir la elección a MACHO y HEMBRA.
* **Familia (abstract):** esta interfaz representa una familia de ratones y define métodos comunes para procrear, comprobar si una hembra es estéril y crear un ratón hijo. Hay tres clases que implementan esta interfaz:
  + **FamiliaNormal**: representa una familia en la cual el macho no tiene ninguna patología. Solo tendrá un ratón madre.
  + **FamiliaPoligamica**: representa una familia en la cual el macho es propenso a la poligamia
  + **FamiliaMachoEsteril:** representa una familia de ratones en la cual el macho es estéril. Puede haber una o varias hembras.
* **ComparatorRatonByNacimiento:** clase que implementa la interfaz Comparator<T> y que permite ordenar ratones por su fecha de nacimiento.
* **ComparatorRatonPesoDescendente:** clase que implementa la interfaz Comparator<T> y que permite ordenar ratones por su peso, de mayor a menor.

# Comprobaciones de integridad y excepciones

## Entrada de datos por consola:

Para manejar los errores en la entrada de datos por teclado se decidió crear 3 métodos input:

* Input(): que utiliza un BufferReader para leer el texto introducido por pantalla y devuelve una cadena de texto (String).
* inputInt(): llama al método input y convierte la cadena de texto a un número entero. Si el valor introducido no es un número entero, saca un mensaje de error vuelve a pedirlo de manera recursiva.
* inputFloat(): sigue la misma lógica que inputInt pero para números reales.

Se decidió aplicar la recursividad en estos métodos para facilitar que inmediatamente después de introducir un valor por teclado, se validase si su tipo es correcto y se volviese a pedir en caso contrario.

No se realizaron más métodos para más tipos de datos primitivos (ej. Double, char, ect.) porque se trabaja principalmente con enteros, reales y Strings.

## Pruebas de validación:

Algunos de los valores que debe introducir el usuario tienen un rango de opción limitado. Ya sea por el formato en el que deben ser introducidos o por un valor mínimo o máximo. Por ejemplo:

* El número de días que permanecerán los ratones en la instalación no puede ser mayor que 630, ni un número negativo. Además, debe ser un múltiplo de 5.
* Para indicar el sexo solo se acepta introducir una M, y H mayúscula.
* Para indicar si un cromosoma estará mutado o no solo se acepta introducir s o n.
* La fecha de nacimiento debe introducirse en el formato yyyy-MM-dd (año-mes-dia) para que el constructor de LocalDate pueda reconocerlo.
* No se puede crear un ratón hembra con un cromosoma Y o un ratón macho sin un cromosoma X y otro Y.
* El peso de un ratón deberá estar entre 0 y 50 gramos para ser válido.
* La temperatura de un ratón deberá estar entre 20º y 40º para ser válido.
* Siempre que se quiera acceder a una población y a sus valores, se comprobará primero que esta existe y que no está vacía.
* Si al construir una población virtual se introducen porcentajes mayores de 100 o menores que 0, no serán válidos.
* No se podrá agregar a una familia varios machos.

En los casos del sexo, mutación y fecha de nacimiento, primero se realiza una comprobación sobre los valores introducidos. En el caso de no ser válidos, se pide al usuario que los vuelva a introducir (de manera recursiva) hasta que el valor introducido sea válido. Estos métodos son bloqueantes. No se continuará con el flujo del programa hasta que dichos valores cumplan los requisitos.

El resto de las comprobaciones se realizan principalmente en los constructores de las clases ratón, población de ratones y familia. Estas lanzarán una excepción en el caso de no ser válidos.

# Estructuras de datos

Para almacenar durante la ejecución del programa los ratones, poblaciones y familias se utilizaron diferentes estructuras de datos:

* TreeMap: Para almacenar los ratones dentro de cada población y familia, además de para almacenar desde el main todas las poblaciones abiertas, se decidió utilizar un TreeMap. Esta estructura permite mantener los elementos ordenados y no permite elementos repetidos. De esta forma no se podrán almacenar 2 ratones con el mismo código de referencia, ni dos poblaciones con el mismo nombre. Además, los mapas pemiten guardar los datos en parejas (clave, valor). De esta forma, permite un acceso más rápido a los valores si se conoce la clave. Se decidió usar esta funcionalidad porque en muchas ocasiones se pedía al usuario introducir el código de referencia o nombre del ratón o población sobre el que se quería trabajar. Ya se conocía la clave. Y acceder al valor era más eficiente utilizando estos valores como claves que iterando por todos los elementos hasta obtener el buscado. (Se es consciente de que al buscar por la clave internamente el mapa ya está iterando)
* LinkedList: para almacenar localmente las familias al generar la simulación se decidió usar una linked list. Como las únicas acciones que se realizarían sobre la lista de familias es insertar y pasar por todos los elementos, se eligió una estructura más eficiente insertando valores.
* TreeSet: Esta estructura se utilizó para crear localmente una copia de los ratones de una población y ordenarlos con criterios de ordenación diferentes.
* Stack: Esta estructura se utilizó para almacenar localmente las hembras y los machos de una población al realizar la simulación de Montecarlo. Las pilas ofrecían métodos propios para sacar el primer elemento de la colección y eliminarlo de la misma. De esa forma, al crear una familia, se sacan de la lista el macho y las hembras que conforman dicha familia para que no puedan formarse más de familias con los mismos progenitores y para seguir creando familias siempre y cuando las dos listas estén llenas.
* Array: los cromosomas de los ratones y los porcentajes calculados al realizar la simulación de Montecarlo se almacenaron en arrays porque permitía el acceso aleatorio a los datos. Además, se utilizaron arrays locales en otras ocasiones en las que no se necesitaba memoria dinámica, el tamaño del array estaba predeterminado, y era necesario el acceso aleatorio.

# Técnicas de ordenación y búsqueda

* Ordenación: para mantener los ratones ordenados y sacarlos por pantalla con diferentes criterios de ordenación se utilizaron estructuras de tipo árbol que de por sí mantienen los datos ordenados.
* Búsqueda: para buscar ratones o poblaciones dentro de las listas se utilizaron los métodos propios de los TreeMap: containsKey(Object Key), get(Object Key).

# Posibles mejoras

**Mejoras:**

* Avisar al usuario de que el ratón que ha creado no se ha podido guardar porque tiene el mismo código de referencia que otro ratón, y permitirle cambiar el código de referencia. De igual forma con las poblaciones.
* Generar de manera aleatoria los códigos de referencia en vez de cargarlos por teclado.
* Mejorar la estética visual de la interfaz gráfica

# Imagen que contiene Escala de tiempo Descripción generada automáticamenteDiagrama de clases UML

# Conclusiones

Para realizar esta práctica se invirtió aproximadamente entre 40-50 horas.

# Bibliografía

*UML Class Diagrams*. (s.f). Recuperado el 26 de 5 del 2023, de Utsa.edu: http://www.cs.utsa.edu/~cs3443/uml/uml.html

‌