Programación III - COMISIÓN 02 - Universidad de General Sarmiento

- Trabajo Práctico 3 -

1er. Cuatrimestre 2023

Profesores

Daniel Bertaccini - Gabriel Carrillo

Alumnos

Cristian Pereyra López - Lautaro Lombardi - Jeremías Ale Carbajales

INTRODUCCIÓN

En el presente trabajo práctico planteamos una posible solución para el análisis e implementación del problema dado como consigna para el Trabajo Práctico 3. En el mismo intentamos utilizar todos los conceptos y herramientas aprendidos en clase con fines utilitarios y pragmáticos para la mejor resolución de todos los problemas que fuimos encontrando en su desarrollo.

Procuramos utilizar y sacar provecho en todo momento de los pilares de la POO que vimos en materias anteriores como encapsulamiento y abstracción, y tuvimos en cuenta a la hora de la implementación por sobre todo el concepto de *"separated presentation"*, es decir, el código que se utiliza para la interfaz debe estar bien encapsulado y separado del código de negocio que ejecuta las funciones esenciales del juego. Nos ajustamos a la arquitectura de diseño de 3 capas (vista-negocios-datos) separando en GUI (Graphical User Interface) lo referido a la parte visual, Logic lo pertinente a lógica de negocio y dal (Data Access Layer) a lo correspondiente a la solicitud tanto en APIs o persistencia de datos.

Para este trabajo además entran en juego los conceptos de los algoritmos de Fuerza Bruta y Backtracking, la implementación de algunos tipos de heurísticas (golosa en nuestro caso) y la utilización de threads para garantizar el poder seguir utilizando el programa mientras este está resolviendo lo pedido con alguno de estos algoritmos. No está de más mencionar la utilización de Swing para la realización de la GUI.

PRESENTACIÓN – EL EQUIPO IDEAL

Nos encontramos en el departamento de recursos humanos de una *software factory*, y tenemos que diseñar un grupo de desarrollo para un nuevo proyecto.

Tenemos una lista de personas, cada una con un rol posible (que puede ser “líder de proyecto”, “arquitecto”, “programador” o “tester”) y una calificación histórica por su desempeño que es un número entre 1 y 5 (cuanto más grande es el número, mejor calificada está la persona). Se cuenta con una lista de pares de personas que en el pasado han tenido inconvenientes al trabajar juntas. Si dos personas están en esta lista, decimos que son incompatibles. Tenemos además una cantidad de personas en cada rol (por ejemplo, necesitamos un líder de proyecto, dos arquitectos, 4 programadores y 5 testers).

La aplicación debe encontrar un conjunto lo más cualificado posible de personas que cumpla estos requerimientos, y que no contenga ningún par de personas incompatibles.

La aplicación debe contar con la siguiente funcionalidad:

- Cargar y visualizar las personas disponibles.
- Cargar y visualizar las incompatibilidades.
- Cargar y visualizar los requerimientos para el equipo.
- Resolver el problema y visualizar el equipo resultante.

Se debe resolver el problema por medio de un algoritmo de fuerza bruta o backtracking, que debe ejecutar en un thread separado de la aplicación principal. Si es posible, se busca como objetivo opcional implementar la resolución de las dos maneras, incluir resoluciones que utilicen Heurísticas, además de una gráfica comparativa entre soluciones y fotos para los empleados.

IMPLEMENTACIÓN

A partir de nuestra elección como arquitectura de diseño un modelado de *"separated presentation"* se optó por realizar una división del código en tres paquetes: por un lado tenemos un paquete de GUI con la interfaz de usuario que contiene la clases pertinentes a las distintas ventanas del juego, otro paquete de la lógica con las clases relativas a la lógica de negocio en donde se encuentra una clase que gestiona los algoritmos para el funcionamiento del core de la aplicación, y por último un paquete de Datos en donde se ubican las clases que interviene en el guardado y recuperación de datos de los empleados y equipos generados.

Visual /IGU

Decidimos crear tres ventanas;

HomeScreen: el inicio en donde el usuario ve los roles de empleados para armar el equipo y puede indicar cuantos de cada uno necesitará en su conformación.Todos los campos deben ser completados o un aviso visual se mostrará como alerta.

MainScreen: la ventana principal de resolución del programa, incluye una lista de empleados precargada y una de conflictos, además de poseer dos secciones para agregar nuevos empleados (que se identifican con su DNI) y nuevos conflictos que no se hayan cargado, además de una serie de botones para buscar una solución con diversos métodos de forma individual y una para resolverlo con todas para ejecutar una comparativa. También desde esta ventana se puede disparar la ***Employee Screen*****:** ventana con forma de identificación que emerge al hacer doble click sobre un empleado y que muestra sus datos personales.

ComparisonScreen: ventana emergente que se abre si se ejecuta la opción para encontrar el equipo con todos los métodos a la vez, muestra tres tablas con la configuración de empleados resultante de cada algoritmo, otra con datos comparativos de cada algoritmo ejecutado y un graficó *jFreeChart* para destacar visualmente la exactitud y diferencia entre los ratings promedios resultantes de la solución de cada algoritmo. Además, la clase del *Observer* correspondiente a esta que es notificado cuando un nuevo equipo o comparativa son generados.

También en esta capa tenemos las clases referentes a *SwingWorkers* de cada algoritmo que permiten seguir utilizando la visual mientras se corren sus procesos

Lógica de Negocio/Logic

Aquí tenemos una clase principal que se ocupa de la entrada y gestión de los elementos de esta capa llamada *idealTeam,* la cual creamos bajo el patrón de diseño *Singleton*, en la cual también incluimos el patrón *Strategy* para la inyección del *comparator* al algoritmo de heurística y un sistema de notificación a *Observers* gracias a los cuales se informa la generación de un nuevo equipo a la visual y a la capa de datos. Además una clase abstracta *Algorithm* de la cual heredan *Bruteforce, Backtracking y Heuristic* (esto nos permitió hacer un código más reducido, reutilizable y con posibilidad de ampliarse) los cuales utilizan métodos de recursión para ejecutar su proceso de búsqueda.

Data Access Layer / DAL

En este paquete ubicamos lo pertinente al almacenamiento y recuperación de los empleados con sus datos y los conflictos entre ellos y los equipos generados. Las listas de empleados y sus conflictos se encuentran en dos archivos *.json*, y hay dos clases de carga y guardado de datos que implementan interfaces para desacoplar y hacer nuestro código reutilizable. La clase *saveData* es notificada mediante un *Observer* y guarda en un *worklog* los equipos generados junto a sus datos (hora de ejecución, tiempo del proceso, rating promedio) y si se ejecutó una comparativa, crea un nuevo archivo .txt con los equipos generados por cada algoritmo y los datos de cada uno.

La interacción entre las tres capas es la adecuada al fin del funcionamiento de la aplicación, y acotado a lo respectivo de cada una. Intentamos utilizar en nuestra diagramación y codificación los patrones de diseño aprendidos.

PROBLEMAS ENCONTRADOS

En los trabajos anteriores solíamos enumerar una lista con varios problemas de mayor o menor gravedad. Pero en este en particular utilizaremos esta sección para mencionar un problema en particular que, si bien no es tan complejo, nos tuvo ocupados por una semana, dado que más allá de este problema concreto no experimentamos problemas serios/dignos de mención.

Cada vez que se ejecuta algún método de resolución, se reemplaza mientras esté en ejecución el botón por una barra de carga.

Tras programar los métodos de resolución, "conectar" su ejecución a cada botón y chequear que su funcionalidad era correcta, nos encontramos con un problema curioso cuanto menos, y era que, tras abrir el programa, si bien a la primera ejecución de cada botón, el programa procedía y encontraba el equipo, cuando se trataba de ejecutar un método que ya había sido ejecutado por 2da vez el programa entraba en loop y el botón se quedaba con la animación de carga eternamente. El programa seguía funcionando lo más normal dentro de sus posibilidades, podías seguir agregando personas y conflictos y no tiraba ninguna excepción en consola.

Entendimos que, si el programa se ejecutaba la primera vez, el problema debía estar en la orden de ejecución del thread o en los propios threads (lo cual descartamos porque cada botón tiene un thread individual y no creíamos que se hubieran programado mal los 3 a la vez).

No encontrábamos alguna pista que nos dijera donde estaba el problema, corríamos mentalmente las zonas del código una y otra vez pero no veíamos ninguna cosa fuera de lugar.

Durante la clase del 23/5, estuvimos hablando del problema, y mientras hacíamos un debugging minucioso línea por línea el código mientras leiamos la documentación en Eclipse de cada instrucción de swingworker y descubrimos que, cuando el *Worker* finaliza su operación y se cierra, este no puede volver a ser ejecutado nuevamente (se debe inicializar un nuevo *Worker*). Esto nos llevó a mirar el código del *MainScreen* (que es de donde se llama al *Worke*r una vez clickeado el botón) y pudimos notar finalmente el error: ***cuando la ventana se creaba, se asociaba el Worker a cada botón sólo cuando recién se creaba y no cada vez que se lo clickeaba, esto llevaba a que la primera vez se ejecutara el Worker normal y se cerrara, pero en la 2da, al estar el Worker inicial cerrado (y sabiendo que ya no se puede abrir), nunca se ejecutaba la parte lógica porque no había thread que realizara la búsqueda***.
Se procedió a modificar esa parte del código para que cada vez que se clickee el botón se le asocié un worker que ejecute la operación y de esa forma el programa empezó a funcionar correctamente sin loops.

LINK A REPOSITORIO DE GITHUB

https://github.com/cristianpl99/tp_3_equipo_ideal