# Informe de Diseño

Integrantes:
Valentín Duke, Gabriel Ortiz, Philippe Pefaur

---

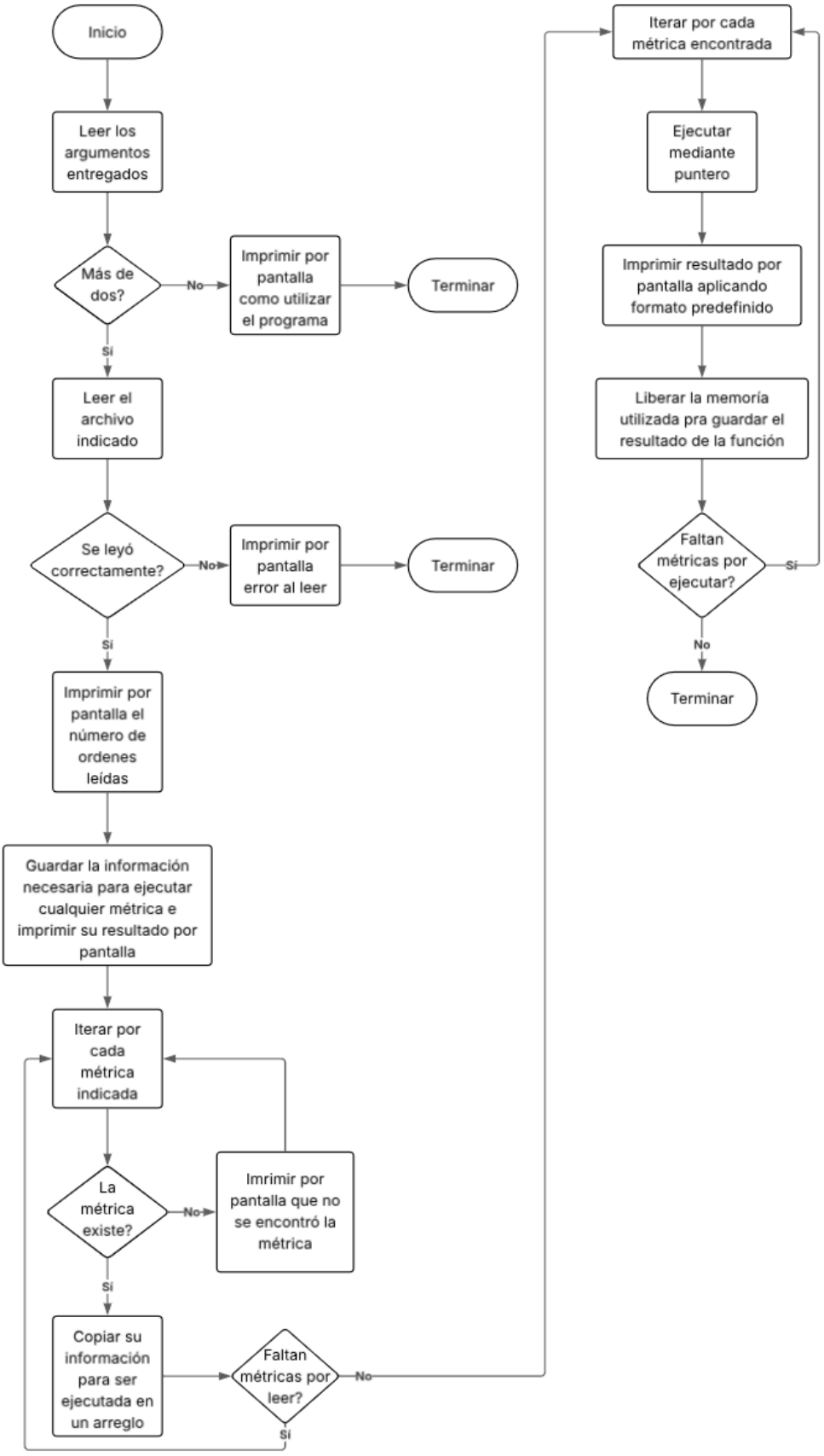
**Objetivo:**

El código utiliza dos estructuras de datos, primero, *struct order* que guarda la información de las órdenes que se encuentran en el archivo .csv indicado. Segundo, *struct metricas* que guarda la información necesaria para llamar (mediante punteros a funciones) e imprimir por pantalla el resultado de las funciones de métricas, esta se tuvo que crear para poder modificar el texto impreso por pantalla para cada una sin incluirlo en el return de cada función.

Además, la aplicación está compuesta por tres archivos de código fuente llamados, *main.c*, donde se encuentra únicamente la función *main()*. Luego, está el archivo *metrics.c*, donde se encuentran las funciones que analizan todas las *struct order* creadas tras leer el archivo .csv y las utilizan para calcular las métricas que fueron indicadas en los requisitos funcionales del programa, esto significa que se encuentran 10 funciones en este archivo, cada una nombrada como se observó en las instrucciones. Por último, en *utils.c* se encuentran las funciones que fueron utilizadas para realizar los procedimientos escritos en los demás archivos que no corresponden a funciones de métricas, estas son, *read_csv()* para leer el archivo .csv indicado y guardar su información en los atributos correspondientes de múltiples variables *struct order* (una por cada orden en el .csv) y la función *conteo_por_nombre()* que registra el número de pizzas según su nombre.

Finalmente, en cuanto al uso de punteros a funciones, estos son aplicados en *main.c* para llamar sólamente las funciones de métricas indicadas por el usuario y en el orden que las escribió. Con mayor detalle, esto se logra creando un arreglo de 10 *struct metricas* (número fijo) que guarda el puntero a cada métrica y su formato para imprimir el resultado por pantalla, luego se itera por cada argumento indicado por el usuario después del nombre del archivo, este se compara con cada posible nombre y, al encontrar una coincidencia, se guarda en el primer espacio vacío de otro arreglo (inicialmente vacío) de *struct metricas* la métrica indicada utilizando el arreglo fijo de 10 *struct metricas*, luego, se itera por cada métrica en este arreglo inicialmente vacío y se llama la función utilizando el puntero, se guarda la dirección a su resultado, se imprime por pantalla y se libera la memoria en la dirección.

**Diagrama de flujo:**



**Razones de diseño:**

Primero, cabe explicar el sistema de ejecución de las métricas. En un principio, estas fueron escritas para retornar únicamente una dirección de memoria dinámica donde se guardó sólo el valor de la métrica, por ejemplo, par pms sólo se guardaba el nombre de la pizza más vendida, esto significaba que, cuando se imprimiera el resultado por pantalla habría que añadir texto y formato para hacer la presentación más amigable al usuario. Ahora bien, cuando se escribió main para poder ejecutar las funciones con sus punteros se encontraron grandes problemas a la hora de aplicar este formato sin hacer una cadena de if else para cada métrica. Por tanto, para poder hacer una iteración que ejecute las funciones sin usar condicionales, se optó por guardar el puntero a las funciones junto a su formato permitiendo en cada iteración acceder a la información adecuada sin condicionales. Es por culpa de este método de implementación que antes de leer los argumentos indicados por el usuario se debe crear un arreglo de *struct_metricas* para tener la información necesaria para ejecutar cada métrica lista para copiar en el arreglo que contiene las funciones que sí se deben ejecutar, además, este método, por su naturaleza, conserva el orden en el que se pidieron las funciones en la ejecución de estas.

Luego, en cuanto al método de parseo del archivo .csv, la función se guiaba por la estructura definida anteriormente y tomaba los strings del csv según su posición para luego asignarlo a una categoría. Por ejemplo, el primer término, hasta la primera coma se etiquetaba como un id de pizza. Ahora bien, se encontró un problema al leer los ingredientes de las órdenes con este método, ya que, estos estaban escritos como una lista acotada por “” que separaba los ingredientes con *,* lo que provocaba que el parseo definiese los ingredientes como sólo el primer ingrediente (más la “ que le precede) y luego el nombre de la pizza como el resto de la línea (ya que terminaba al encontrar un salto de línea \n). Para resolver esto se separó a estos dos atributos del procedimiento general y se trataron por separado, utilizando las “” para identificar el inicio y final de los ingredientes y guardandolos como un único string que contiene todos los ingredientes separados por *,*, luego, se utiliza el índice donde se encontró la última comilla para conocer el índice donde se encuentra la primera letra del nombre de la pizza y así guardarla apropiadamente.

**Explicación de la interacción entre archivos:**

El programa está dividido en diferentes archivos, data_structure.h contiene la organización de los datos que se extraigan del csv, en donde se definen que tipo de dato (char, float, etc) serán cada una de las columnas o cabezales dentro del csv. A continuación, los archivos utils.c y utils.h corresponden a la función de lectura del csv y una función que se utiliza en las métricas pms y pls para evitar procedimientos repetitivos, donde el primero de estos archivos contiene la lógica de estas, y el segundo la firma de las funciones para que otros archivos la puedan ocupar dentro del proyecto. Continuando, los archivos metric.c y metric.h, corresponden a las funciones de las métricas pedidas, siendo el primero la lógica y el segundo la firma. Finalmente, tenemos los archivos main.c y Makefile, el primero integra todas las funciones de los archivos anteriores y ejecuta cada una de las funciones de métricas

dependiendo del input que se pida y el segundo funciona como compilador de todo el proyecto para que este pueda ser ejecutable desde la terminal.

**Reflexiones finales:**

La principal complejidad enfrentada durante el desarrollo de este proyecto fué la complejidad del código pedido (utilizando punteros a funciones y estructuras de datos, memoria dinámica y teniendo que trabajar constantemente con strings de caracteres) y especialmente la modularidad de este, lo cual provocó confusión múltiples veces durante la implementación y debugging de las funciones más allá de lo complejo que podía llegar a ser entender los errores que se encontraban debido a los punteros y strings. Entonces, para poder resolver estos problemas fue que se utilizaron chats de IA para asesorar el desarrollo permitiendo encontrar rápidamente información necesaria para resolver los errores encontrados y realizar la modularidad correctamente, además de preguntar al profesor en clase por dudas menos urgentes. Con esto se logró aprender rápidamente acerca de los requisitos funcionales del programa para controlar los conceptos detrás de ellos y desarrollar un programa de alta complejidad sin sufrir gran presión por culpa de la falta de tiempo.

**Uso de IA:**

El uso de IA depende de cada integrante, por lo que se listara la experiencia de cada uno individualmente:

1. Valentín Duke:

   Por mi lado, soy consciente de que soy principiante en la escritura con lenguaje C, y se tuvo que aprender a cómo este lenguaje se estructura y como los archivos dentro de un proyecto en C interactúan. Además, fue la primera vez que se trabajó con un archivo Makefile, no se sabía bien la función de estos ni cómo se implementan en un proyecto. De esta forma, la IA agilizó este proceso y pudimos aprender cosas avanzadas rápidamente.

   De todos modos, es importante aclarar que las respuestas de la IA eran estrictamente fiscalizadas. Sabemos que las inteligencias generativas se pueden equivocar, por eso es que toda las soluciones que nos dio las estudiamos y comprobamos que sean correctas.

2. Gabriel Ortiz:

   Este integrante utilizó GTP 4o para analizar el código ya escrito y que el chat otorgara sugerencias de código para escribir algunas funciones del programa, luego las modificó levemente para que fueran más coherentes con el resto del proyecto y, en caso de que hubiese algún problema de implementación utilizaba el chat como asesor para comprender el error y así resolver el conflicto.

3. Philippe Pefaur:

Este integrante utilizó Copilot de Github con Claude 3.5 Sonnet integrado en Visual Studio Code para consultar acerca de los errores o avisos que entregaba el compilador durante el desarrollo del programa. En estos casos, el chat respondía con una explicación comprensiva del error y mostrando una modificación del código que resolvería el problema, ahora bien, esta sugerencia la gran mayoría de veces provocaría nuevos errores o avisos o incluso modificaría la lógica de secciones del código haciéndolas incompatibles con el programa en general por lo que no era efectivo simplemente copiarla. Luego, las respuestas que entregaba el chat eran utilizadas para resolver el error manualmente ahora que sí se entendía (gracias a la explicación comprensiva del error) por qué razón el segmento de código problemático estaba provocando errores.