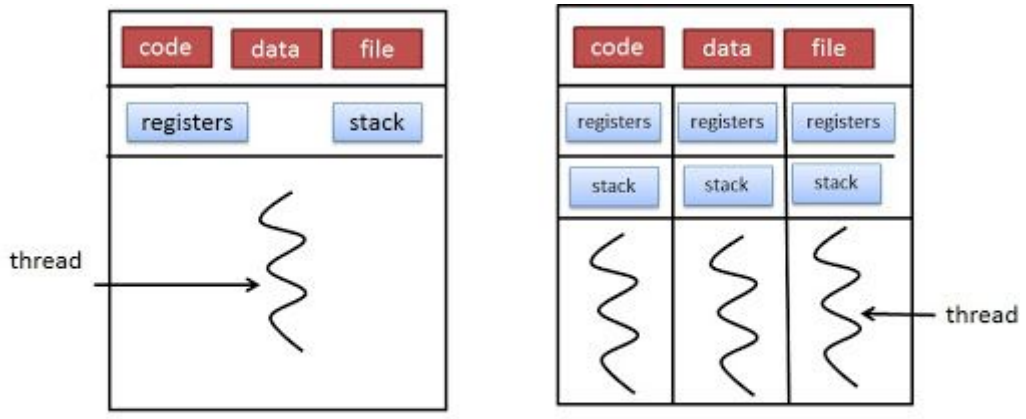

Single-threaded Process Multithreaded Process

Programación Concurrente

Trabajo Práctico de Laboratorio Nº1
Productores y Consumidores

Docentes:

- Dr. Ing. Micolini Orlando
- Ing. Ventre, Luis
- Ing. Ludemann, Mauricio

Autores:

Matrícula	Apellido y Nombre	Carrera
41279796	Bonino, Francisco Ignacio	Ing. en Comp.
40502859	Coronati, Federico Joaquín	Ing. en Comp.
39129465	Luna, Lihué Leandro	Ing. en Comp.
41232347	Merino, Mateo	Ing. en Comp.

Grupo Code Busters
Año 2020

Índice

Objetivo

El objetivo de este trabajo práctico de laboratorio es el de tener un primer acercamiento a la **programación concurrente** y utilizar los conocimientos adquiridos en clase para realizar un programa de productores y consumidores poniendo especial atención en el comportamiento de los hilos durante la ejecución del programa.

El **enunciado** del trabajo se presenta a continuación:

En un buffer cuya capacidad son 25 lugares, acceden múltiples consumidores y múltiples productores (es decisión del grupo la cantidad de consumidores y de productores). Este buffer tiene pura pérdida, es decir que si está lleno, el productor descarta los elementos. Cada consumidor puede extraer solo de a un elemento por vez y demora un tiempo aleatorio en milisegundos para consumirlo. Cada productor puede añadir solo de a un elemento por vez y demora un tiempo aleatorio en milisegundos para generarlo. El sistema posee un log que escribe cada 2 segundos, y tiene por objetivo realizar una estadística del uso del buffer y de los estados de los consumidores. El log debe registrar:

a) *Cantidad de lugares ocupados del buffer.*

b) *Estado de cada consumidor (ocupado consumiendo, o disponible).*

Ejercicios:

1) *Hacer un diagrama de clases que modele el sistema del buffer con los productores y consumidores.*

2) *Hacer un solo diagrama de secuencias que muestre la siguiente interacción:*

   a) *Con el buffer vacío, un productor coloca un producto satisfactoriamente*

   b) *Luego, dos consumidores llegan al mismo tiempo a consumir. Mostrar qué pasa con cada consumidor.*

3) *Modelar el buffer, los productores y los consumidores, con objetos en Java. Se deben generar y consumir 1000 elementos en total.*

4) *Explicar los motivos de los resultados obtenidos.*

5) *Debe hacer una clase Main que al correrla, inicie el programa.*

6) *Los tiempos de generación y consumo de elementos en el buffer, deben configurarse para que el programa demore entre 60 segundos y 120 segundos en finalizar.*

Diagramas en UML

Como primera medida se plantearon los **diagramas de clase y de secuencia en UML**. Sin embargo, por la poca experiencia del equipo de trabajo en lo que respecta a ingeniería de software, se dificultó este paso y se hicieron unos diagramas sencillos antes de codificar para tenerlos como croquis, sujetos a posteriores cambios.

Luego de codificar en Java basándose en los primeros diagramas en UML, se decidió modificarlos con las ideas que surgieron en la implementación del código; en donde se modificaron tanto las clases a utilizar como los métodos empleados, dando lugar al diagrama de clases que ilustra lo implementado en el programa y las relaciones entre cada sección del código.

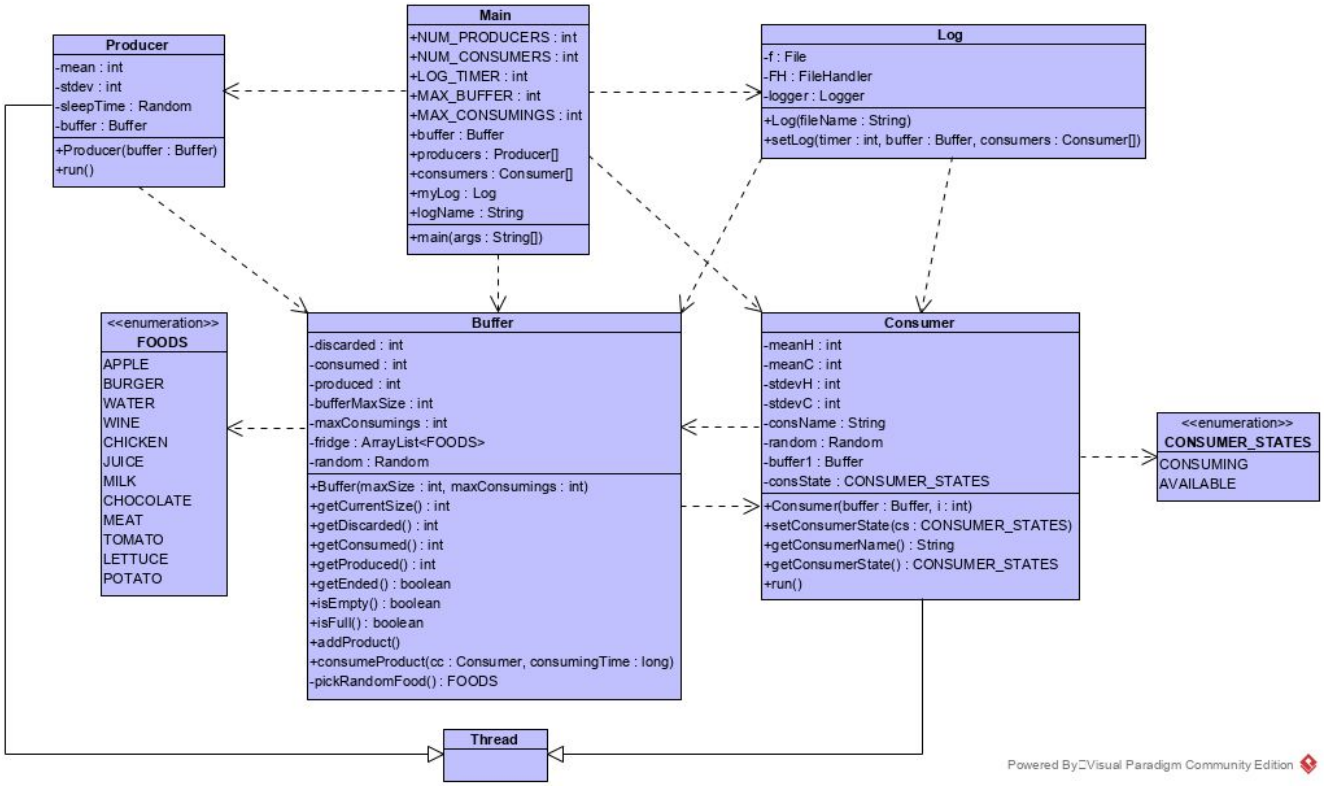

Figura 1: Diagrama de Clases

Una vez listo el diagrama de la Figura 1, se procedió a construir el diagrama de secuencia según el **Lenguaje Unificado de Modelado (UML)**. La Figura 2 ilustra el diagrama mencionado, el cual muestra la situación hipotética en la que dos consumidores quieren consumir al mismo tiempo, uno de ellos demora un tiempo en consumir mientras que el otro espera a que termine el primero, además de chequear que haya productos disponibles.

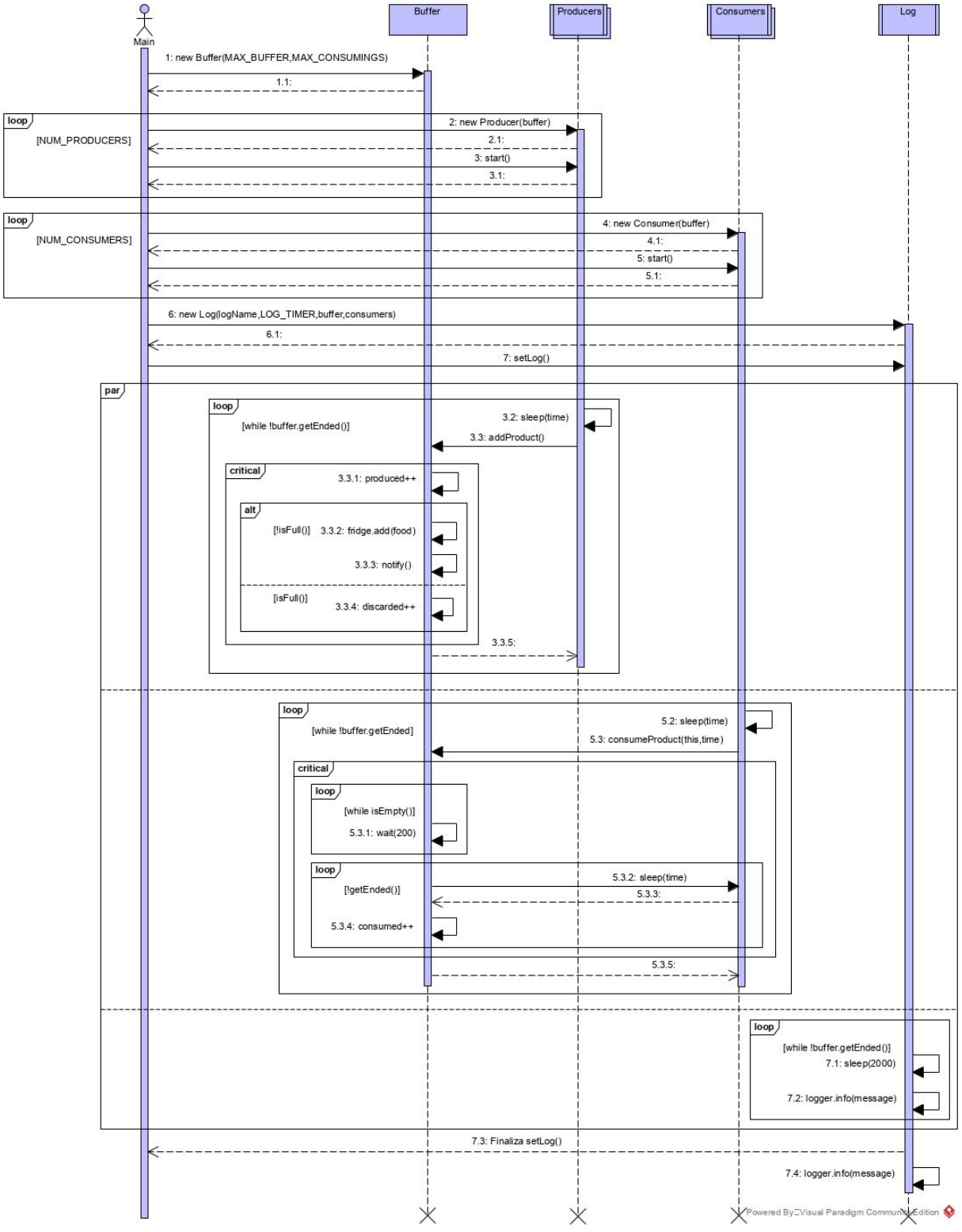

Figura 2: Diagrama de Secuencia

Implementación en Java

En cuanto a la implementación en Java, se tomaron varios caminos posibles para abordar el problema. A continuación se comenta brevemente la funcionalidad de cada clase y sus métodos asociados con los que el grupo resolvió lo pedido.

- **Main**: Clase principal. Se encargará de crear las instancias de las demás clases y comenzar la ejecución de los hilos.

- **Buffer**: Esta clase está destinada a trabajar como *contenedor* para los objetos. En este caso, el buffer simula ser una heladera. Para representar los alimentos que se almacenarán en nuestro buffer, se creó un enumerador con comidas varias. Cada vez que productor agregue algo, esto será un elemento tomado del enumerador mencionado, de forma aleatoria con distribución de probabilidad uniforme.

- **Consumidor**: La clase encargada de consumir los productos que luego se almacenarán en el buffer, lo cual toma un tiempo aleatorio en milisegundos con distribución normal, para que los hilos *"tengan hambre"* y otro tiempo de distribución normal para consumir.

- **Productor**: Clase encargada de generar los productos del buffer. Esta operación demora un tiempo aleatorio en milisegundos, con distribución normal.

- **Log**: Esta clase se ocupa de generar el objeto que escribirá en un archivo de texto cada 2 segundos los espacios ocupados del buffer y el estado de cada hilo consumidor junto a su fecha y hora, para poder hacer los testeos con cierto grado de formalismo. Para definir los estados se creó un enumerador que define los posibles estados: *consumiendo* y *disponible*.

Los **métodos** más importantes de cada una de las clases se describen brevemente a continuación:

*Clase **Producer:***

- **run()**: Este método es una sobreescritura del método run() de la clase *Thread*. Se usa para describir el comportamiento del productor, el cual duerme un momento (el tiempo que le lleva producir el producto) y llama a *addProduct()* para agregarlo. Este comportamiento se repite hasta consumir 1000 unidades.

*Clase **Consumer:***

- **run()**: Sobreescritura del método *run()* de la clase *Thread*. El consumidor duerme un tiempo *("hasta que le de hambre")* y llama a c*onsumeProduct()* indicando en su argumento el tiempo que le costará consumir el producto. De manera análoga, este comportamiento también se repite hasta consumir 1000 unidades.

*Clase **Buffer:***

- **isEmpty()**: Indica si el buffer tiene cero elementos o no.
- **isFull()**: Indica si el buffer tiene 25 elementos o no.
- **getEnded()**: Devuelve ***true*** si se consumieron 1000 productos.
- **getCurrentSize()**: Devuelve la cantidad de productos que hay en el buffer en ese momento.
- **addProduct()**: Intenta agregar un producto al arraylist que representa la heladera. Si ésta está llena, se descarta el producto.
- **consumeProduct()**: Intenta consumir un elemento de la heladera. Si ésta se encuentra vacía, el hilo espera; sino, quita un producto *(lo consume)*.
- **pickRandomFood()**: Devuelve una posible comida a añadir a la heladera; determinada aleatoriamente mediante una variable random.

*Clase **Log:***

- **setLog():** Imprime en pantalla cada 2 segundos el estado de cada consumidor. Cuando finaliza el programa, imprime un diagnóstico con los datos que se almacenaron en el buffer durante la ejecución.

Durante la implementación, se llevaron a cabo varios pasos en conjunto con el equipo de trabajo, los cuales condujeron a algunos problemas como es frecuente en la implementación de software. Estos problemas se describen a continuación.

En primer lugar se optó por **sincronizar bloques** de instrucciones para reducir la sección crítica y de esta manera optimizar los tiempos de ejecución. Estos bloques sincronizados se encontraban en las clases *Producer* y *Consumer*, de manera tal que el programa en sí funcionaba correctamente; pero no contemplaba que lo que realmente debe estar sincronizado son los métodos para agregar y quitar productos en la clase *Buffer*, así interpretándose correctamente que tanto

productores como consumidores ingresan a una *(en nuestro caso)* “heladera”; justamente donde se encuentran los recursos compartidos *(secciones críticas)*.

Se decidió entonces **sincronizar métodos** (*addProduct()* y *consumeProduct(), ambos métodos pertenecientes a la clase Buffer*) para productores y consumidores respectivamente, teniendo éstos únicamente un tiempo de *sleep* en su método *run()* antes de solicitar el acceso a sus secciones críticas, las cuales están dentro de métodos sincronizados, por lo que nos aseguramos que serán ejecutados de a un hilo a la vez.

Una vez solucionada la sincronización, surgieron otros problemas: en principio, al querer acceder a el estado de un hilo consumidor, el método ***getState()*** devolvía los estados propios de la clase *Thread,* es decir: *running, waiting,* etc. Para corregir esto, se modificó la clase *Consumer* para que en lugar de implementar la interfaz *Runnable*, extienda la clase *Thread*, sobreescribiendo su método ***run()***. Esta es una manera alternativa de crear hilos, sólo difiere en que en la clase *Main,* en lugar de crear objetos tipo *Thread* con argumento *runnable* de tipo *Consumer*, se crea directamente un objeto *Consumer* que hereda de *Thread* con su respectivo método *run().* A partir de esta modificación se creó un enumerador *ConsumerStates* para definir 2 posibles estados de los consumidores: ***AVAILABLE*** y ***CONSUMING.*** Estos serán asignados a un atributo de la clase *Consumer* llamado *consState*, que será particularmente útil a la hora de documentar en el log.

Se investigó la forma de crear un **log** o **bitácora de trabajo**, que se realizó creando una clase llamada ***Log*** e importando las clases *Logger*, *FileHandler* y *SimpleFormatting* del paquete *java.util.logging*.

Finalmente se corrieron las pruebas pertinentes para asegurar el correcto funcionamiento del programa en función de lo solicitado, y se agregaron los comentarios correspondientes para comprender mejor la ejecución del programa. Los tiempos elegidos para las variables normales que se utilizan al dormir a los hilos se seleccionaron para que la ejecución del programa en su totalidad demore entre 1 y 2 minutos, según el hardware que lo ejecute.

Conclusión

Durante la realización del presente trabajo práctico, se tomaron ciertas decisiones de diseño para cumplir el objetivo pactado.

Se decidió que tanto los productores como los consumidores tuvieran una **distribución de probabilidad normal**, ya que esta distribución describe adecuadamente los procesos que tienden a acercarse a un valor medio con cierta aleatoriedad. Los valores adoptados para media y desviación estándar se seleccionaron a criterio, pensando en que generalmente se tarda más en producir que en consumir. Incluso se agregó un tiempo de *"hambre"*, es decir, que los consumidores sientan necesidad de consumir al cabo de un tiempo y no cuando recién terminan de consumir un producto. Al equipo de trabajo le pareció que este comportamiento simula mejor un sistema de productores y consumidores. Los valores elegidos para las medias y desvíos estándares son:

Magnitud	**Valor**
Media de Producción	150 [ms]
Desvío Estándar de Producción	15 [ms]
Media de Consumición	70 [ms]
Desvío Estándar de Consumición	15 [ms]
Media de Hambre	110 [ms]
Desvío Estándar de Hambre	15 [ms]

Además de los valores de tiempo escogidos, se seleccionaron diferentes cantidades de hilos productores y consumidores para ver el comportamiento del sistema. Se concluyó por seleccionar **10 productores** y **6 consumidores** para que ocurra el desecho de productos y se pueda aplicar la siguiente ecuación para verificar el buen funcionamiento del software.

$$Producidos = Consumidos + Desechados + Restantes\ en\ el\ Buffer$$

Se verificó la igualdad anterior con varias ejecuciones del código cambiando las variables de trabajo y en todos los casos se cumple.

Gracias a la realización de este trabajo práctico, como grupo nos dimos cuenta de que la programación concurrente es de difícil comprensión y muchas veces toma comportamientos anti intuitivos que dificultan la resolución de los problemas, sin embargo la eficiencia temporal que se alcanza al ejecutar las tareas de manera simultánea es significativa y reconocemos que es un campo que vale la pena aprender, para crear programas cada vez más rápidos y que estén a la altura de las necesidades contemporáneas.