

Universidad Latina de Costa Rica

BIT-22 Sistemas Operativos – II Cuatrimestre 2022

Profesor

Carlos Andrés Méndez Rodríguez

Estudiante

Oscar Andrés Pinto Villegas

2007011502

Tema

“Implementación en Java de uno de los problemas clásicos de comunicación entre procesos (IPC) – Problema de los Filósofos Comelones”

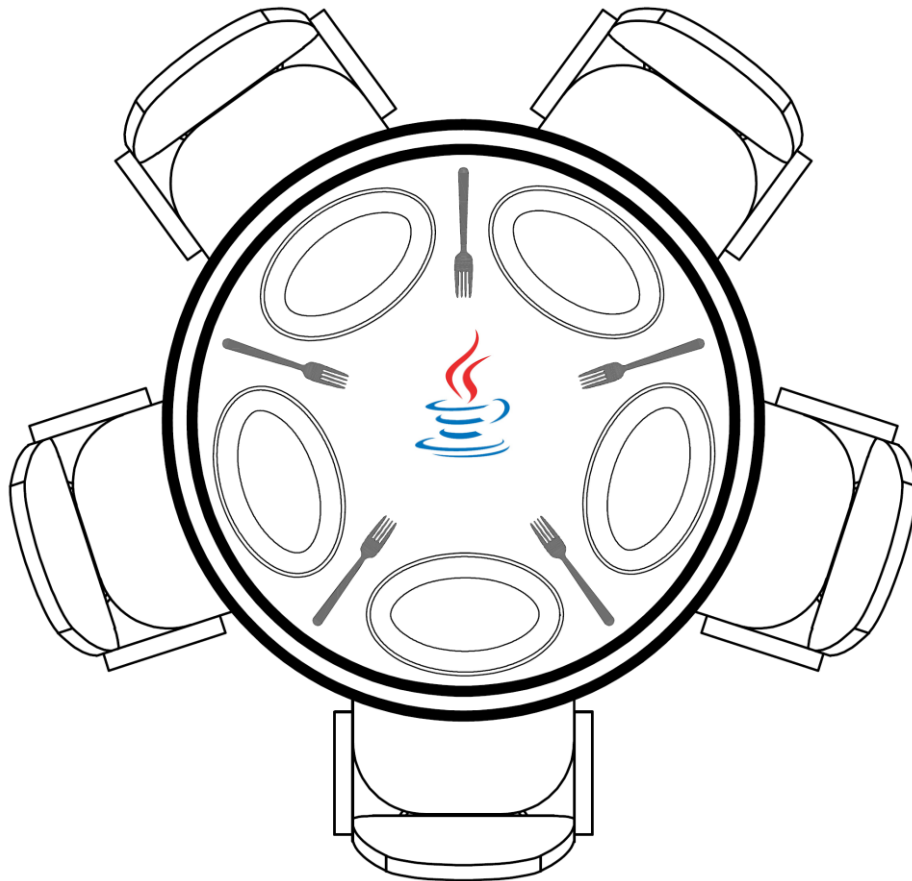


Tabla de contenido

Solicitud del Proyecto	3
Tema	3
Entregables	3
¿Qué es Java?	4
¿Qué es una API?	4
Breve historia de las API	4
¿Cómo funcionan las API?	5
¿Qué son las integraciones de las API?	6
¿Cuáles son los diferentes tipos de API?	6
API Java	6
Thread.....	7
Clase Thread.....	7
Métodos de la Clase Thread	7
Métodos de Instancia de la Clase Thread	8
Creación de un Thread.....	9
Clase MemoryUsage	10
Paquete java.lang.management.....	12
MBean	12
MXBean	12
Plataforma MXBean.....	12
ManagementFactory	13
ThreadMXBean	13
El Problema de los Filósofos Comelones	13
Resolución del Problema de los Filósofos Comelones en Java	14
Microservicios y una Implementación Conceptual al Problema de los Filósofos Comelones	23
¿Qué es una Arquitectura Monolítica?	23
¿Qué son los Microservicios?.....	24
Bibliografía.....	28

SOLICITUD DEL PROYECTO

Para el curso *Sistemas Operativos* el profesor Carlos Andrés Méndez Rodríguez brindo el documento “*ENUNCIADO PROYECTO.pdf*” en el cual se indica lo siguiente:

“(…) Se asignará 1 proyecto que fomente la investigación, estudio y otros para alcanzar los objetivos planteados en el trabajo en clase y reforzar los temas vistos en clase. (…)”

Tema

Considerando lo solicitado por el profesor se decidió realizar una implementación en el lenguaje de programación Java de uno de los problemas clásicos de comunicación entre procesos (IPC por sus siglas en inglés, *InterProcess Communication*). El problema seleccionado es el “*Problema de los Filósofos Comelones*” (mayormente conocido por su nombre en inglés “*The Dining Philosophers Problem*”).

En razón de lo anterior, el proyecto se denominó:

“Implementación en Java de uno de los problemas clásicos de comunicación entre procesos (IPC) – Problema de los Filósofos Comelones”

Entregables

En el documento “*ENUNCIADO PROYECTO.pdf*” se solicitan una serie de entregables, los cuales se enumeran a continuación:

“(…) Proyecto de programación

El(los) estudiante(s) pueden diseñar, analizar, implementar y probar uno o varios programas que les sea de interés, debe cumplir lo siguiente:

- 1. Se puede utilizar cualquier lenguaje de programación y/o una combinación.*
- 2. Deberá implementar, no solo diseñar, analizar y probar.*
- 3. Deberá revisar la API de la tecnología utilizada y profundizar en su comprensión y uso. (…)*
- 4. Se pueden poner en práctica también otras librerías para memoria, planificación, archivos de entrada/salida, sistemas distribuidos y seguridad.*
- 5. No se requiere trabajar en una máquina virtual. (…)*

Importante: Si el estudiante no tiene registros en Linux o GitHub que muestren implementación obtendrá un cero en todos los rubros anteriores. (…)”

¿Qué es Java?¹

Java es una plataforma informática de lenguaje de programación, ampliamente utilizada para codificar aplicaciones web, creada por *Sun Microsystems* en 1995. Ha sido una opción popular entre los desarrolladores durante más de dos décadas, con millones de aplicaciones *Java* es un lenguaje multiplataforma, orientado a objetos y centrado en la red que se puede utilizar como una plataforma en sí mismo. Es un lenguaje de programación rápido, seguro y fiable para codificar todo, desde aplicaciones móviles y software empresarial hasta aplicaciones de macrodatos y tecnologías del lado del servidor.

¿Qué es una API?²

Las **API** son mecanismos que permiten a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos.

Por ejemplo, un sistema de software de un instituto de meteorología contiene datos meteorológicos diarios; la aplicación meteorológica de un teléfono inteligente “habla” con este sistema a través de las **API** y muestra las actualizaciones meteorológicas diarias en dicho dispositivo.

El término **API** es una abreviatura de *Application Programming Interfaces* (traducido al español como *Interfaz de Programación de Aplicaciones*). En el contexto de las **API**, la palabra aplicación se refiere a cualquier software con una función distinta. La interfaz puede considerarse como un contrato de servicio entre dos aplicaciones. Este contrato define cómo se comunican entre sí mediante solicitudes y respuestas. La documentación de una **API** contiene información sobre cómo los desarrolladores deben estructurar esas solicitudes y respuestas.

Dicho esto, es posible decir que una **API** es una especificación formal que se establece cómo un módulo de un software, el cual se comunica o interactúa con otro para cumplir una o muchas funciones. En razón de las aplicaciones que se vayan a utilizar, y de los permisos que el propietario de la **API** les proporcione a los desarrolladores.

Breve historia de las API³

Las **API** surgieron en los comienzos de la informática, mucho antes que la computadora personal. En esa época, una **API** normalmente se usaba como biblioteca para los sistemas operativos. Casi siempre estaban habilitadas localmente en los sistemas en los que operaban, aunque a veces pasaban mensajes entre las computadoras centrales. Después de casi 30 años, las **API** se expandieron más allá de los entornos locales. A principios del año 2000, ya eran una tecnología importante para la integración remota de datos.

¹ Información extraída de los sitios web <https://aws.amazon.com/es/what-is/Java/> y https://www.java.com/es/download/help/whatis_Java.html.

² Información extraída del sitio web <https://aws.amazon.com/es/what-is/api/>.

³ Información extraída del sitio web <https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces>.

¿Cómo funcionan las API?⁴

La arquitectura de las **API** suele explicarse en términos de cliente y servidor. La aplicación que envía la solicitud se llama cliente, y la que envía la respuesta se llama servidor. Considerando el ejemplo mencionado anteriormente, la *base de datos meteorológicos del instituto* es el servidor y la *aplicación móvil* es el cliente.

Las **API** pueden funcionar de cuatro maneras diferentes, según el momento y el motivo de su creación:

- **API de SOAP**, estas **API** utilizan el *Protocolo Simple de Acceso a Objetos (SOAP)*, por sus siglas en inglés *Simple Object Access Protocol*). El cliente y el servidor intercambian mensajes mediante *XML*⁵. Se trata de una **API** menos flexible que era más popular en el pasado.
- **API de RPC**, estas **API** se denominan *Llamadas a Procedimientos Remotos (RPC)*, por sus siglas en inglés *Remote Procedure Call*). El cliente completa una función (o procedimiento) en el servidor, y el servidor devuelve el resultado al cliente.
- **API de WebSocket**, la **API de WebSocket** es otro desarrollo moderno de la **API** web que utiliza objetos *JSON*⁶ para pasar datos. La **API de WebSocket** admite la comunicación bidireccional entre las aplicaciones cliente y el servidor. El servidor puede enviar mensajes de devolución de llamada a los clientes conectados, por lo que es más eficiente que la **API de REST**.
- **API de REST**⁷, estas son las **API** más populares y flexibles que se encuentran en la web actualmente. **API REST** es una **API** que se ajusta a los principios de diseño de *REST*, un estilo de arquitectura también denominado *Transferencia de Estado Representacional*. Por este motivo, las **API REST** son a veces denominadas **API RESTful**. En ellas el cliente envía las solicitudes al servidor como datos y el servidor utiliza esta entrada del cliente para iniciar funciones internas y devuelve los datos de salida al cliente.

⁴ Información extraída del sitio web <https://aws.amazon.com/es/what-is/api/>.

⁵ *XML*, siglas en inglés de *eXtensible Markup Language*, traducido como “*Lenguaje de Marcado Extensible*” o “*Lenguaje de Marcas Extensible*”. *XML* es un lenguaje de marcado, al igual que el *HTML* (utilizado para programar páginas Web), definido y mantenido por el *World Wide Web Consortium (W3C)*. El objetivo del *XML* se enfoca en la simplicidad, generalidad y usabilidad por parte de toda la Internet. Aunque el *XML* apunte en particular a la generación de documentos, también se lo utiliza para representar estructuras de datos arbitrarias, apuntando a su integración entre sistemas de computadores.

⁶ *JavaScript Object Notation (JSON)* es un formato basado en texto estándar para representar datos estructurados en la sintaxis de objetos de JavaScript. Es comúnmente utilizado para transmitir datos en aplicaciones web (por ejemplo: enviar algunos datos desde el servidor al cliente, así estos datos pueden ser mostrados en páginas web, o vice versa). Aunque es muy similar a la sintaxis de objeto literal de JavaScript, puede ser utilizado independientemente de JavaScript, y muchos entornos de programación poseen la capacidad de leer (convertir; parsear) y generar *JSON*.

⁷ Información extraída del sitio web <https://www.ibm.com/mx-es/cloud/learn/rest-apis>.

¿Qué son las integraciones de las API?⁸

Las integraciones de las **API** son componentes de software que actualizan automáticamente los datos entre los *clientes* y los *servidores*.

Algunos ejemplos de integraciones de las **API** son la sincronización automática de datos en la nube desde la galería de imágenes de un teléfono inteligente o la sincronización automática de la hora y la fecha en una computadora portátil cuando se viaja a otra zona horaria, o también las empresas pueden utilizarlas para automatizar de manera eficiente muchas funciones de su sistema.

¿Cuáles son los diferentes tipos de API?⁹

Las **API** se clasifican tanto en función de su arquitectura como de su ámbito de uso. Dentro de su ámbito de uso se encuentran:

- **API Privadas**, estas son internas de una empresa y solo se utilizan para conectar sistemas y datos dentro de la empresa.
- **API Públicas**, están abiertas al público y pueden ser utilizadas por cualquier persona. Puede haber o no alguna autorización y coste asociado a este tipo de **API**.
- **API de Socios**, solo pueden acceder a ellas los desarrolladores externos autorizados para ayudar a las asociaciones entre empresas.
- **API Compuestas**, estas combinan dos o más **API** diferentes para abordar requisitos o comportamientos complejos del sistema.

API Java

La **API Java** es una interfaz de programación de aplicaciones provista por los creadores del lenguaje de programación Java, que da a los programadores los medios para desarrollar aplicaciones dicho lenguaje. Como el lenguaje Java es un lenguaje orientado a objetos, la **API Java** provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa. Además, la **API Java** está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente.

Una vez descrito que es una **API**, cuál es su función, cuáles son sus diferentes tipos, etc; se llegó a la conclusión que para resolver el problema propuesto es posible con la **API Java**, específicamente asiendo uso de la clase **Thread** que se encuentra en ella.

Antes de definir qué en que consiste la clase **Thread**, es importante mencionar algunos conceptos, entre ellos que es un **Thread** y en que consiste.

⁸ Información extraída del sitio web <https://aws.amazon.com/es/what-is/api/>.

⁹ Ibidem.

Thread

Los **Threads**, también conocidos como **Hilos de ejecución** o simplemente **Hilos** (por su traducción al español o *Hebras* en algunas literaturas); son una secuencia de tareas encadenadas, muy pequeña, que puede ser ejecutada por un sistema operativo. Otras literaturas definen a un **Thread** como una característica que permite a una aplicación realizar varias tareas a la vez (concurrentemente).

Los **Threads** son una ampliación del concepto de *Multitarea* (*Multitasking*); el concepto de *Multitarea* se refiere a la capacidad de un sistema para ejecutar varios *procesos*¹⁰ a la vez, se dice que en un comienzo esto hacía referencia a que más de una aplicación se estuviese ejecutando de manera *concurrente*, sin embargo, pronto se hizo notoria la necesidad de que una misma aplicación hiciera varias cosas a la vez. De allí nacieron los **Threads**.

Los **Hilos de Ejecución** (**Threads**) que comparten los mismos recursos, sumados a estos recursos, son en conjunto conocidos como un *proceso*. El hecho de que los **Threads** de un mismo *proceso* compartan los recursos hace que cualquiera de estos **Threads** pueda modificar estos recursos. Cuando un **Thread** modifica un dato en la memoria, los otros **Threads** acceden a ese dato modificado inmediatamente. Lo que es propio de cada **Thread** es el contador de programa, la pila de ejecución y el estado de la CPU (incluyendo el valor de los registros). El *proceso* sigue en ejecución mientras al menos uno de sus **Threads** de ejecución siga activo. Cuando el *proceso* finaliza, todos sus **Threads** también han terminado. Asimismo, en el momento en el que todos los **Threads** finalizan, el proceso no existe más y todos sus recursos son liberados.

Clase Thread

Java Thread es una de las clases más clásicas del **API Java** y es la encargada de ejecutar tareas en paralelo creando nuevos **Threads**. Un **Java Thread** está diseñado apoyándose en el concepto de composición y permite recibir una tarea que implemente la interfaz **Runnable**.

La clase **Thread**, es la clase que encapsula todo el control necesario sobre **Threads** (**Hilos de Ejecución**). Hay que distinguir claramente un objeto Thread de un *Hilo de Ejecución* o **Thread**. Esta distinción resulta complicada, aunque se puede simplificar si se considera al objeto Thread como el panel de control de un **Thread**. La clase **Thread** es la única forma de controlar el comportamiento de los **Threads** y para ello posee ciertos métodos.

Métodos de la Clase Thread

Los métodos estáticos que deben llamarse de manera directa en la clase **Thread**, son:

- **currentThread()**. Este método devuelve el objeto Thread que representa al **Hilo de Ejecución** que se está ejecutando actualmente.

¹⁰ Los *procesos* son una de las abstracciones más antiguas e importantes que proporcionan los sistemas operativos: proporcionan la capacidad de operar (pseudo) concurrentemente, incluso cuando hay sólo una CPU disponible. Convierten una CPU en varias CPU virtuales.

- **yield()**. Este método hace que el intérprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible. Es una manera de asegurar que los hilos de menor prioridad no sufran *inanición*¹¹.
- **sleep(long)**. El método *sleep()* provoca que el intérprete ponga al hilo en curso a dormir durante el número de milisegundos que se indiquen en el parámetro de invocación. Una vez transcurridos esos milisegundos, dicho hilo volverá a estar disponible para su ejecución. Los relojes asociados a la mayor parte de los intérpretes de Java no serán capaces de obtener precisiones mayores de 10 milisegundos, por mucho que se permita indicar hasta nanosegundos en la llamada alternativa a este método.

Métodos de Instancia de la Clase Thread

A continuación, se detallan algunos métodos de la *Clase Thread*, porque los demás métodos corresponden a áreas en donde el estándar de Java no está completo, y puede que se queden obsoletos en las siguientes versiones del *JDK (Java Development Kit)*, por ello, para completar la información expuesta se ha de recurrir a la documentación del API del JDK.

- **start()**. Este método indica al intérprete de Java que cree un contexto del hilo del sistema y comience a ejecutarlo. A continuación, el método *run()* de este hilo será invocado en el nuevo contexto del hilo. Hay que tener precaución de no llamar al método *start()* más de una vez sobre un hilo determinado.
- **run()**. El método *run()* constituye el cuerpo de un hilo en ejecución. Este es el único método del interfaz *Runnable*. Es llamado por el método *start()* después de que el hilo apropiado del sistema se haya inicializado. Siempre que el método *run()* devuelva el control, el hilo actual se detendrá.
- **stop()**. Este método provoca que el hilo se detenga de manera inmediata. A menudo constituye una manera brusca de detener un hilo, especialmente si este método se ejecuta sobre el hilo en curso. En tal caso, la línea inmediatamente posterior a la llamada al método *stop()* no llega a ejecutarse jamás, pues el contexto del hilo muere antes de que *stop()* devuelva el control. Una forma más elegante de detener un hilo es utilizar alguna variable que ocasione que el método *run()* termine de manera ordenada. En realidad, nunca se debería recurrir al uso de este método.
- **suspend()**. El método *suspend()* es distinto de *stop()*. *suspend()* toma el hilo y provoca que se detenga su ejecución sin destruir el hilo de sistema subyacente, ni el estado del hilo anteriormente en ejecución. Si la ejecución de un hilo se suspende, puede llamarse a *resume()* sobre el mismo hilo para lograr que vuelva a ejecutarse de nuevo.

¹¹ En informática, *inanición* (*starvation* en inglés) es un problema relacionado con los sistemas multitarea, donde a un proceso o un hilo de ejecución se le deniega siempre el acceso a un recurso compartido. Sin este recurso, la tarea a ejecutar no puede ser nunca finalizada.

- **resume()**. El método *resume()* se utiliza para revivir un hilo suspendido. No hay garantías de que el hilo comience a ejecutarse inmediatamente, ya que puede haber un hilo de mayor prioridad en ejecución actualmente, pero *resume()* ocasiona que el hilo vuelva a ser un candidato a ser ejecutado.
- **setPriority(int)**. El método *setPriority()* asigna al hilo la prioridad indicada por el valor pasado como parámetro. Hay bastantes constantes predefinidas para la prioridad, definidas en la **clase Thread**, tales como *MIN_PRIORITY*, *NORM_PRIORITY* y *MAX_PRIORITY*, que toman los valores 1, 5 y 10, respectivamente. Como guía aproximada de utilización, se puede establecer que la mayor parte de los procesos a nivel de usuario deberían tomar una prioridad en torno a *NORM_PRIORITY*. Las tareas en segundo plano, como una entrada/salida a red o el nuevo dibujo de la pantalla, deberían tener una prioridad cercana a *MIN_PRIORITY*. Con las tareas a las que se fije la máxima prioridad, en torno a *MAX_PRIORITY*, hay que ser especialmente cuidadosos, porque si no se hacen llamadas a *sleep()* o *yield()*, se puede provocar que el intérprete Java quede totalmente fuera de control.
- **getPriority()**. Este método devuelve la prioridad del hilo de ejecución en curso, que es un valor comprendido entre uno y diez.
- **setName(String)**. Este método permite identificar al hilo con un nombre *mnemónico*¹². De esta manera se facilita la depuración de programas multihilo. El nombre *mnemónico* aparecerá en todas las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.
- **getName()**. Este método devuelve el valor actual, de tipo cadena, asignado como nombre al hilo en ejecución mediante *setName()*.

Creación de un Thread

Hay dos modos de conseguir **Threads** en Java. Una es implementando el interfaz **Runnable**, la otra es extender la **clase Thread**.

La implementación del interfaz **Runnable** es la forma habitual de crear hilos. Los interfaces proporcionan al programador una forma de agrupar el trabajo de infraestructura de una clase. Se utilizan para diseñar los requerimientos comunes al conjunto de clases a implementar. El interfaz define el trabajo y la clase, o clases, que implementan el interfaz para realizar ese trabajo. Los diferentes grupos de clases que implementen el interfaz tendrán que seguir las mismas reglas de funcionamiento. Existen algunas diferencias entre interfaz y clase, estas se resumen a continuación:

1. Un interfaz solamente puede contener métodos abstractos y/o variables estáticas y finales (constantes). Las clases, por otro lado, pueden implementar métodos y contener variables que no sean constantes.

¹² En informática, un *mnemónico* o *nemónico* es una palabra que sustituye a un código de operación (lenguaje de máquina), con lo cual resulta más fácil la programación, es de aquí de donde se aplica el concepto de lenguaje ensamblador.

2. Un interfaz no puede implementar cualquier método. Una clase que implemente una interfaz debe implementar todos los métodos definidos en ese interfaz.
3. Un interfaz tiene la posibilidad de poder extenderse de otros interfaces y, al contrario que las clases, puede extenderse de múltiples interfaces.
4. Un interfaz no puede ser instanciado con el operador *new*, por ejemplo, la siguiente sentencia no está permitida:

```
Runnable a = new Runnable(); // No se permite
```

Otra forma de instanciar la clase **Thread** es por medio de un *pool*. Esto por lo general se hace por practicidad del programador que este desarrollando el código, en el presente documento se mostraran ambos casos.

Clase MemoryUsage

Un objeto **MemoryUsage**¹³ representa “fotografía instantánea” del uso de la memoria. Las instancias de la clase **MemoryUsage** suelen ser construidas por métodos que se utilizan para obtener información sobre el uso de la memoria de cada uno de los pools de memoria de la Java Virtual Machine (**JVM**¹⁴ o *Máquina Virtual Java*) del *heap* o de la *Non-Heap Memory* de la **JVM** en su totalidad. En la **Imagen 1** se muestra un ejemplo de un pool de memoria junto con los cuatro valores de un objeto **MemoryUsage**.

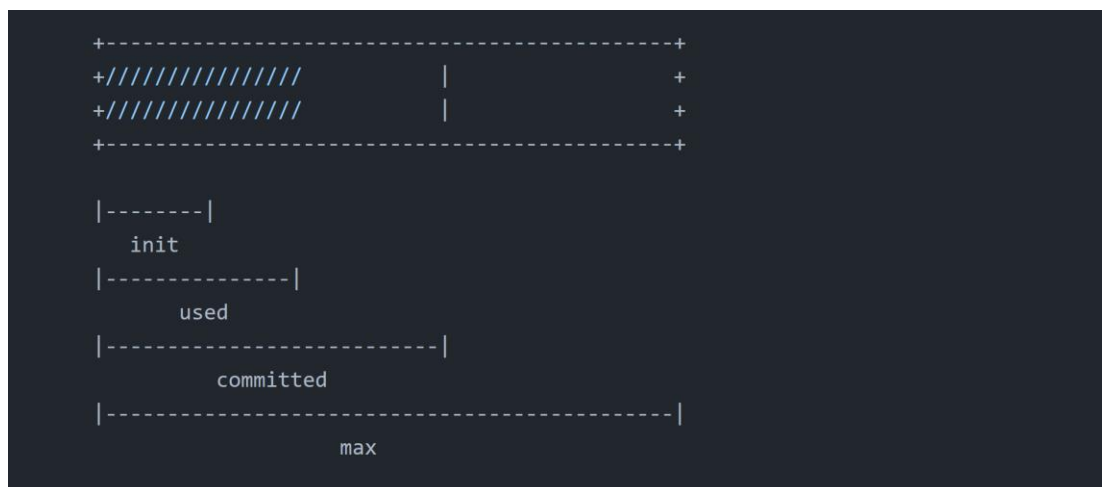


Imagen 1. Representación gráfica de los valores contenidos un objeto **MemoryUsage**.

Los valores del **MemoryUsage** son:

- **init()**. Este valor representa la cantidad inicial de memoria (en bytes) que la Máquina Virtual Java solicita al sistema operativo para la gestión de la memoria durante el arranque. La Máquina Virtual Java puede solicitar memoria adicional

¹³ Extraído del sitio web <https://runebook.dev/es/docs/openjdk/java.management/java/lang/management/memoryusage>.

¹⁴ La Máquina Virtual Java (**JVM**) es el entorno en el que se ejecutan los programas Java, su misión principal es la de garantizar la portabilidad de las aplicaciones Java. Define esencialmente un ordenador abstracto y especifica las instrucciones (bytecodes) que este ordenador puede ejecutar. Extraído del sitio web <http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/introduccion/virtual.htm>.

del sistema operativo y también puede liberar memoria al sistema con el tiempo. El valor de *init* puede ser indefinido.

- ***used ()***. Los valores acá incluidos, representan la cantidad de memoria que se utiliza actualmente (en bytes).
- ***committed ()***. Representa la cantidad de memoria (en bytes) que se garantiza que estará disponible para el uso de la Máquina Virtual Java, esta cantidad de memoria puede cambiar con el tiempo (aumentar o disminuir). La Máquina Virtual Java puede liberar memoria al sistema y *committed* podría ser menor que *init*; pero *committed* siempre será mayor o igual que *used*.
- ***max ()***. Finalmente, representa la cantidad máxima de memoria (en bytes) que se puede utilizar para la gestión de memoria. Su valor puede ser indefinido, esto permite que la cantidad máxima de memoria puede cambiar con el tiempo si se define, adicionalmente la cantidad de memoria utilizada y comprometida siempre será menor o igual a *max* si *max* es definido. Una asignación de memoria puede fallar si intenta aumentar la memoria utilizada de manera que *used* > *committed* incluso si se *used* <= *max* aún sería verdadero.

El sistema de memoria de la Máquina Virtual de Java maneja los siguientes tipos de memoria:

- ***Heap*** – La Máquina Virtual de Java tiene un *heap*; el *heap* corresponde al área de datos *runtime*¹⁵ (tiempo de ejecución) que se asigna para almacenar todas las instancias de clases y los arrays (arreglos), en otras palabras, el *heap* (o almacenamiento libre) es el área de memoria usada para la asignación dinámica de memoria. Este es creado cuando la Máquina Virtual de Java; la memoria *heap* para los objetos es reclamada por un sistema automático de gestión de memoria que se conoce como *garbage collector* (recolector de basura, traducido al español). El *heap* puede poseer un tamaño fijo o puede expandirse y reducirse. La memoria para el *heap* no necesita ser continua.
- ***Non-Heap Memory*** – La Máquina Virtual de Java gestiona otra memoria aparte de la *heap*, esta se llama Non-Heap Memory (Memoria no Acumulativa), además esta misma Máquina Virtual posee un área de método que es compartida entre todos los *threads*. El área de método (o *method area*, en inglés) pertenece a la Non-Heap Memory, acá se almacenan estructuras según su clase como un pool constante de runtime, métodos y campos de datos y el código para los métodos y constructores; esta área se crea cuando la Máquina Virtual de Java arranca. El área del método es lógicamente parte del *heap*, pero una implementación de Máquina Virtual de Java puede optar por no recoger la basura ni compactarla. Al igual que el *heap*, el *method area* puede ser de un tamaño fijo o puede expandirse

¹⁵ *Runtime*, es el periodo en el que un programa comienza a correr, empieza desde que se abre el programa, es decir, cuando el Sistema Operativo comienza a ejecutar sus instrucciones y hasta que se cierra, ya sea de forma normal o porque produjo algún error y el sistema forzó su cierre. Extraído del sitio web <http://www.icorp.com.mx/blog/que-es-el-runtime/>.

y reducirse. La memoria del *method area* no necesita ser contigua. Además del *method area*, una implementación de la Máquina Virtual de Java puede requerir memoria para el procesamiento interno o la optimización, lo que también pertenece a la *Non-Heap Memory*.

Paquete java.lang.management

Proporciona las interfaces de gestión para la supervisión y gestión de la Máquina Virtual Java y de otros componentes en el *runtime* de Java; además permite la monitorización y gestión tanto local como remota de la Máquina Virtual Java en ejecución.

MBean¹⁶

La tecnología ***MBean*** permite a las aplicaciones Java poseer un método para administrar los recursos de una computadora dentro de una aplicación de software. Un ***MBean*** es la representación virtual de Java de un dispositivo o recurso dentro de una computadora. Los ***MBeans*** exponen una interfaz de administración que permite la manipulación de atributos y operaciones funcionales del recurso, lo que permite el monitoreo y la manipulación en tiempo real de los procesos informáticos.

MXBean¹⁷

Un ***MXBean*** es un tipo de ***MBean*** que hace referencia únicamente a un conjunto predefinido de tipos de datos. De esta forma, se asegura de que el ***MBean*** podrá ser utilizado por cualquier *cliente*, incluidos los clientes remotos, sin necesidad de que el *cliente* tenga acceso a las clases específicas del modelo que representan los tipos de los ***MBeans***. Los ***MXBeans*** proporcionan una forma conveniente de agrupar valores relacionados, sin requerir que los *clientes* estén especialmente configurados para manejar los paquetes.

Plataforma MXBean

Una plataforma MXBean es un *bean*¹⁸ gestionado que se ajusta a la especificación de instrumentación JMX¹⁹ y sólo utiliza un conjunto de tipos de datos básicos. Cada plataforma MXBean es una ***PlatformManagedObject*** con un nombre único.

¹⁶ Extraído del sitio web <https://www.netinbag.com/es/internet/what-is-an-mbean.html>.

¹⁷ Extraído del sitio web <https://docs.oracle.com/javase/tutorial/jmx/mbeans/mxbeans.html>.

¹⁸ ***Bean***, es una clase destinada a almacenar una cantidad de datos para un programa. Su fin es encapsular información, para reutilizar código fuente, estructurando el código fuente en unidades lo más sencillas posible.

¹⁹ Java Management Extensions (***JMX***) es la especificación Java que define la arquitectura de operación, que facilita la gestión de aplicaciones y servicios. Esta tecnología permite que los desarrolladores Java integren sus aplicaciones con las soluciones existentes de gestión y operación. Extraído del sitio web <https://www.adictosaltrabajo.com/2005/11/21/jmx/>.

ManagementFactory

La clase *ManagementFactory* es la clase de fábrica de gestión para la plataforma Java. Esta clase proporciona un conjunto de métodos de fábrica estáticos para obtener los *MXBeans* para la plataforma Java para permitir que una aplicación acceda a los *MXBeans* directamente.

ThreadMXBean

Interfaz de gestión específica de la plataforma para el sistema de hilos de la Máquina Virtual Java.

Una vez descritas las herramientas del API de la tecnología a utilizar en la implementación se procede a detallar el problema en cuestión.

El Problema de los Filósofos Comelones

En 1965, Dijkstra²⁰ propuso y resolvió un problema de sincronización al que llamó el *Problema de los Filósofos Comelones*. Este problema se puede enunciar de la siguiente manera; cinco filósofos están sentados alrededor de una mesa circular y cada filósofo tiene un plato de espagueti. El espagueti es tan resbaloso, que un filósofo necesita de dos tenedores para comerlo y entre cada par de platos hay un tenedor. La distribución de la mesa se ilustra en la **Imagen 2**.

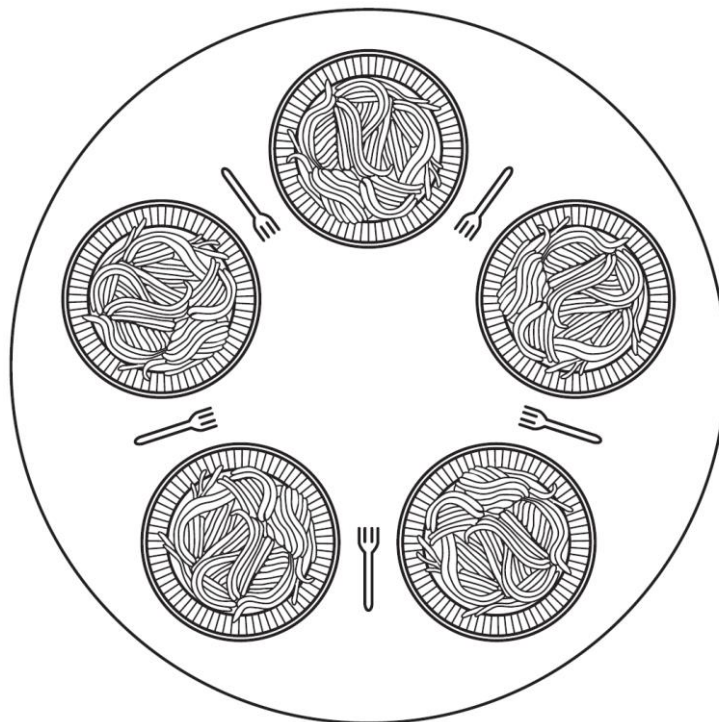


Imagen 2. Distribución de la mesa para el problema de los filósofos Comelones. Imagen extraída del libro Sistemas operativos modernos (3ra ed.).

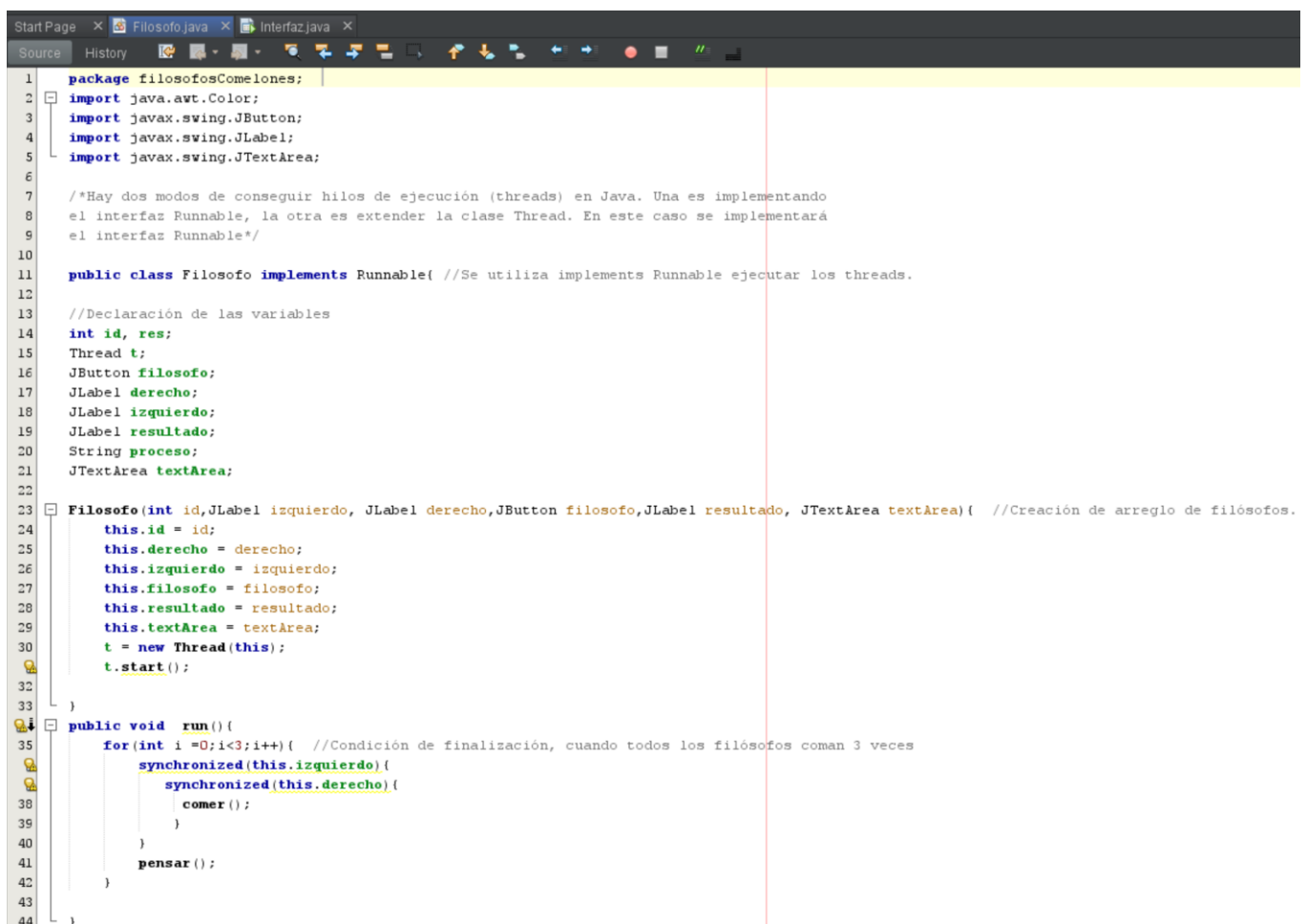
²⁰ Edsger Wybe Dijkstra (Róterdam, 11 de mayo de 1930-Nuenen, 6 de agosto de 2002) fue un científico de la computación de los Países Bajos.

La vida de estos filósofos consiste en periodos alternos de comer y pensar (esto es algo así como una abstracción, incluso para los filósofos, pero las otras actividades son irrelevantes aquí). Cuando un filósofo tiene hambre, trata de adquirir sus tenedores izquierdo y derecho, uno a la vez, en cualquier orden. Si tiene éxito al adquirir dos tenedores, come por un momento, después deja los tenedores y continúa pensando. La pregunta del problema descrito anteriormente es; ¿Cómo se escribiría un programa para cada filósofo, que haga lo que se supone debe hacer y nunca se trabe?

Resolución del Problema de los Filósofos Comelones en Java

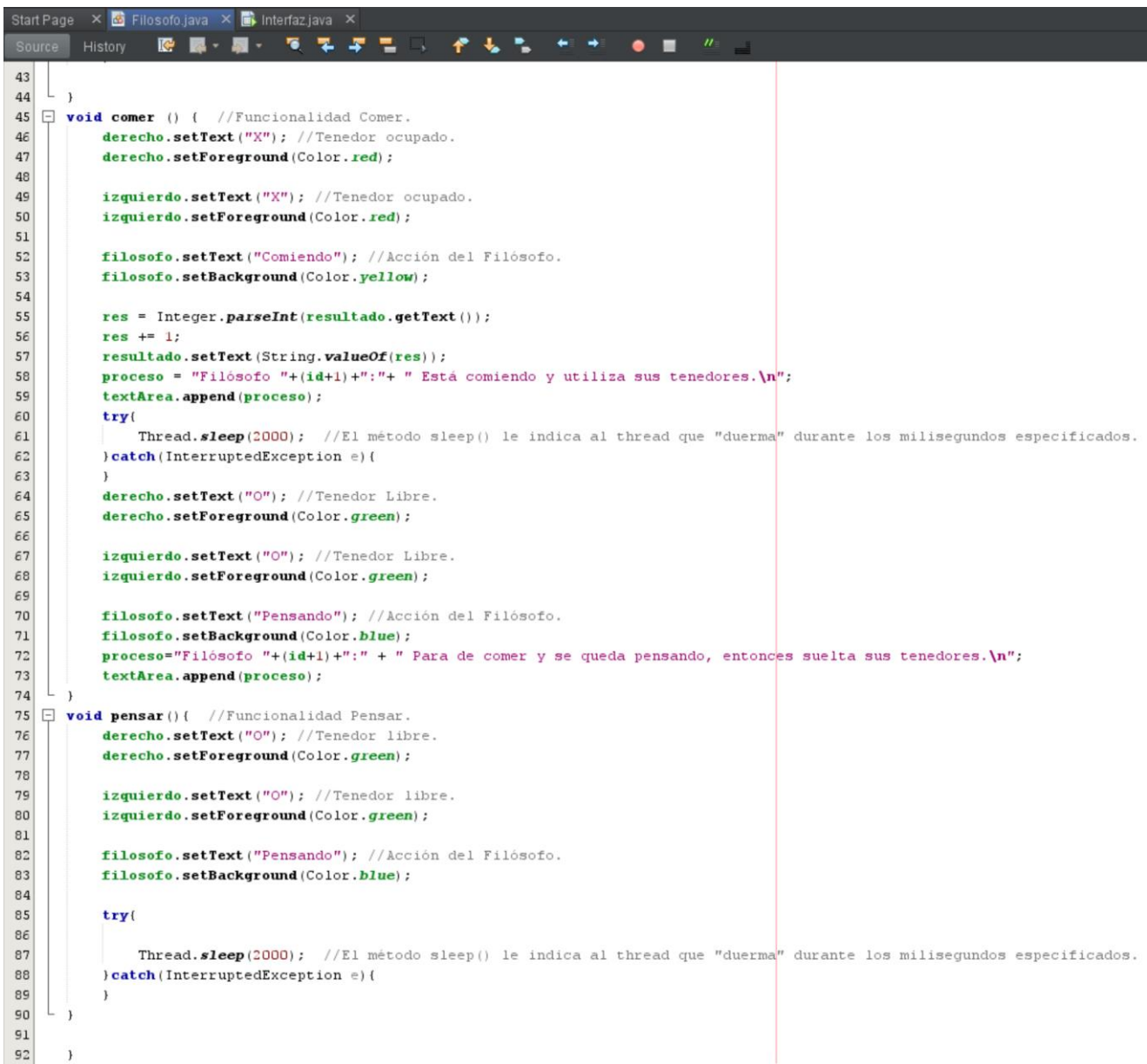
Un programa Java tiene un *Thread* principal con el que este arranca, por cada programa es posible arrancar más *Threads*. Como se mencionó anteriormente, para generar un *Thread* se necesita de objetos que implementen la interfaz **Runnable**; las formas más habituales de generar un *Thread* es implementando la interfaz **Runnable** o extendiendo la *clase Thread* (que a su vez es implementa **Runnable**).

Para la solución propuesta al problema de los filósofos se decidió implementar la interfaz **Runnable**. Para esto inicialmente se utilizó el siguiente código:



```
1 package filosofosComelones;
2 import java.awt.Color;
3 import javax.swing.JButton;
4 import javax.swing.JLabel;
5 import javax.swing.JTextArea;
6
7 /*Hay dos modos de conseguir hilos de ejecución (threads) en Java. Una es implementando
8 el interfaz Runnable, la otra es extender la clase Thread. En este caso se implementará
9 el interfaz Runnable*/
10
11 public class Filosofo implements Runnable( //Se utiliza implements Runnable ejecutar los threads.
12
13 //Declaración de las variables
14 int id, res;
15 Thread t;
16 JButton filosofo;
17 JLabel derecho;
18 JLabel izquierdo;
19 JLabel resultado;
20 String proceso;
21 JTextArea textArea;
22
23 Filosofo(int id, JLabel izquierdo, JLabel derecho, JButton filosofo, JLabel resultado, JTextArea textArea){ //Creación de arreglo de filósofos.
24     this.id = id;
25     this.derecho = derecho;
26     this.izquierdo = izquierdo;
27     this.filosofo = filosofo;
28     this.resultado = resultado;
29     this.textArea = textArea;
30     t = new Thread(this);
31     t.start();
32 }
33
34 public void run(){
35     for(int i =0; i<3; i++){ //Condición de finalización, cuando todos los filósofos coman 3 veces
36         synchronized(this.izquierdo){
37             synchronized(this.derecho){
38                 comer();
39             }
40         }
41         pensar();
42     }
43 }
44 }
```

Imagen 3. Captura de pantalla del código formulado en NetBeans, clase *Filosofo.java*.



```

43 }
44 }
45 void comer () { //Funcionalidad Comer.
46     derecho.setText("X"); //Tenedor ocupado.
47     derecho.setForeground(Color.red);
48
49     izquierdo.setText("X"); //Tenedor ocupado.
50     izquierdo.setForeground(Color.red);
51
52     filosofo.setText("Comiendo"); //Acción del Filósofo.
53     filosofo.setBackground(Color.yellow);
54
55     res = Integer.parseInt(resultado.getText());
56     res += 1;
57     resultado.setText(String.valueOf(res));
58     proceso = "Filósofo "+(id+1)+": "+ " Está comiendo y utiliza sus tenedores.\n";
59     textArea.append(proceso);
60     try{
61         Thread.sleep(2000); //El método sleep() le indica al thread que "duerma" durante los milisegundos especificados.
62     }catch (InterruptedException e){
63     }
64     derecho.setText("O"); //Tenedor Libre.
65     derecho.setForeground(Color.green);
66
67     izquierdo.setText("O"); //Tenedor Libre.
68     izquierdo.setForeground(Color.green);
69
70     filosofo.setText("Pensando"); //Acción del Filósofo.
71     filosofo.setBackground(Color.blue);
72     proceso="Filósofo "+(id+1)+": "+ " Para de comer y se queda pensando, entonces suelta sus tenedores.\n";
73     textArea.append(proceso);
74 }
75 void pensar(){ //Funcionalidad Pensar.
76     derecho.setText("O"); //Tenedor libre.
77     derecho.setForeground(Color.green);
78
79     izquierdo.setText("O"); //Tenedor libre.
80     izquierdo.setForeground(Color.green);
81
82     filosofo.setText("Pensando"); //Acción del Filósofo.
83     filosofo.setBackground(Color.blue);
84
85     try{
86
87         Thread.sleep(2000); //El método sleep() le indica al thread que "duerma" durante los milisegundos especificados.
88     }catch (InterruptedException e){
89     }
90 }
91 }
92 }

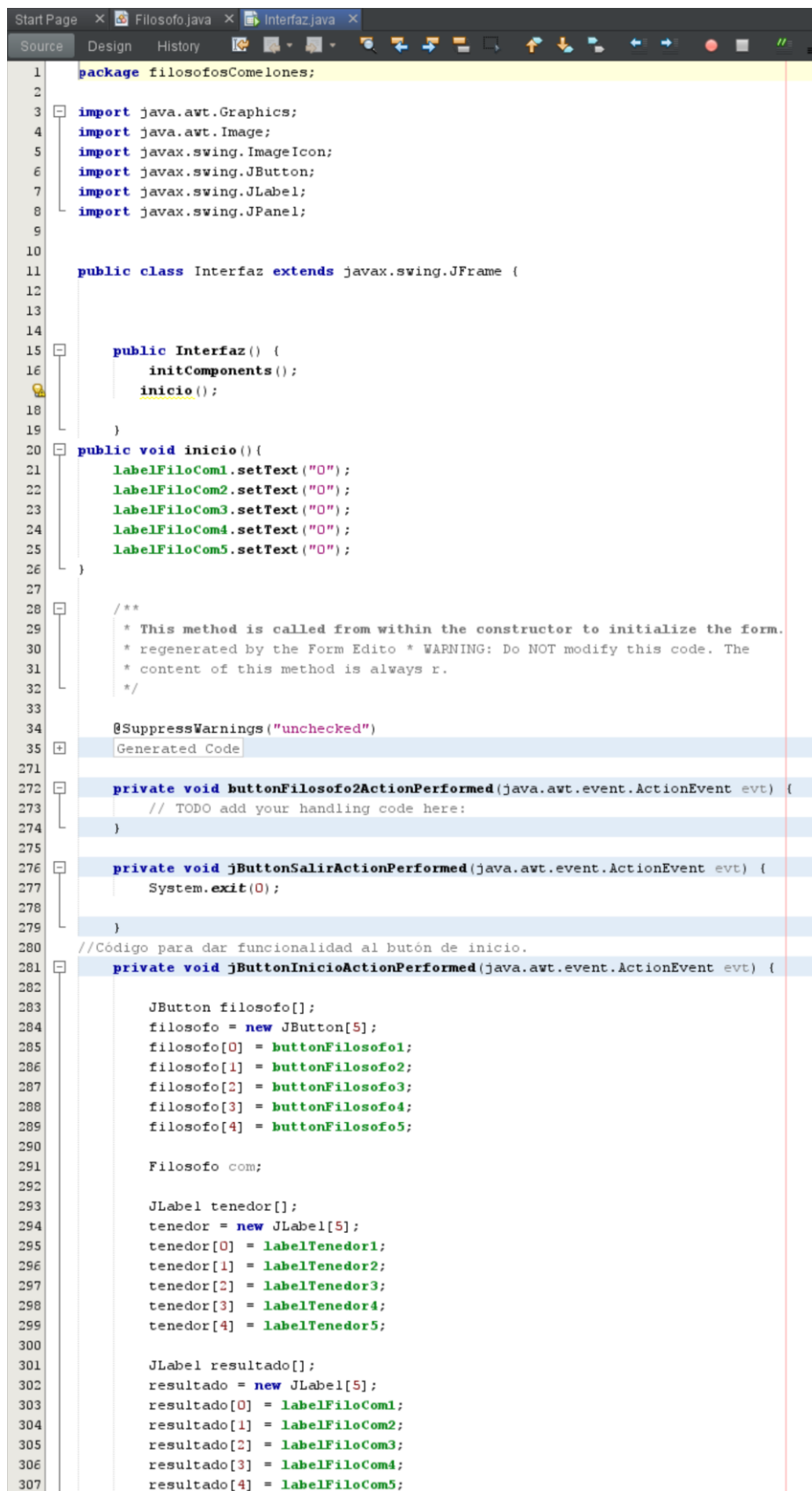
```

Imagen 4. Captura de pantalla del código formulado en NetBeans, clase *Filosofo.java*.

En el código mostrado anteriormente se hace la implementación de la interfaz **Runnable** para ello se creó una clase con nombre *Filosofo.java*. Esta clase incluye las variables propias de los filósofos, así como las acciones que estos ejecutan. Estas acciones se validan o realizan una a la vez según sea el caso, por medio del método aplicado con la palabra reservada **synchronized**²¹.

²¹ **Synchronized** se usa para indicar que ciertas partes del código, (habitualmente, una función miembro) están sincronizadas, es decir, que solamente un subproceso puede acceder a dicho método a la vez. Cada método sincronizado posee una especie de *llave* que puede cerrar o abrir la “puerta de acceso”. Cuando un subproceso intenta acceder al método sincronizado verificará si la *llave* está “puesta”, en cuyo caso no podrá accederlo. Si el método no tiene “puesta” la *llave* entonces el subproceso puede acceder a dicho código sincronizado. Extraído del sitio web <http://www.sc.chu.es/sbweb/fisica/cursoJava/applets/threads/sincronizacion.htm>.

Adicionalmente se creó un JFrame Form de nombre *Interfaz.java* que despliega de forma gráfica el comportamiento de los filósofos, el código es el siguiente:



```
1 package filosofosComelones;
2
3 import java.awt.Graphics;
4 import java.awt.Image;
5 import javax.swing.ImageIcon;
6 import javax.swing.JButton;
7 import javax.swing.JLabel;
8 import javax.swing.JPanel;
9
10
11 public class Interfaz extends javax.swing.JFrame {
12
13
14
15     public Interfaz() {
16         initComponents();
17         inicio();
18     }
19
20     public void inicio(){
21         labelFiloCom1.setText("0");
22         labelFiloCom2.setText("0");
23         labelFiloCom3.setText("0");
24         labelFiloCom4.setText("0");
25         labelFiloCom5.setText("0");
26     }
27
28     /**
29      * This method is called from within the constructor to initialize the form.
30      * regenerated by the Form Editor. * WARNING: Do NOT modify this code. The
31      * content of this method is always r.
32      */
33
34     @SuppressWarnings("unchecked")
35     Generated Code
36
37
38     private void buttonFilosofo2ActionPerformed(java.awt.event.ActionEvent evt) {
39         // TODO add your handling code here:
40     }
41
42     private void jButtonSalirActionPerformed(java.awt.event.ActionEvent evt) {
43         System.exit(0);
44     }
45
46     //Código para dar funcionalidad al botón de inicio.
47     private void jButtonInicioActionPerformed(java.awt.event.ActionEvent evt) {
48
49         JButton filosofo[];
50         filosofo = new JButton[5];
51         filosofo[0] = buttonFilosofo1;
52         filosofo[1] = buttonFilosofo2;
53         filosofo[2] = buttonFilosofo3;
54         filosofo[3] = buttonFilosofo4;
55         filosofo[4] = buttonFilosofo5;
56
57         Filosofo com;
58
59         JLabel tenedor[];
60         tenedor = new JLabel[5];
61         tenedor[0] = labelTenedor1;
62         tenedor[1] = labelTenedor2;
63         tenedor[2] = labelTenedor3;
64         tenedor[3] = labelTenedor4;
65         tenedor[4] = labelTenedor5;
66
67         JLabel resultado[];
68         resultado = new JLabel[5];
69         resultado[0] = labelFiloCom1;
70         resultado[1] = labelFiloCom2;
71         resultado[2] = labelFiloCom3;
72         resultado[3] = labelFiloCom4;
73         resultado[4] = labelFiloCom5;
```

Imagen 5. Captura de pantalla del código formulado en NetBeans, JFrame Form *Interfaz.java*.

```
Start Page x Filosofo.java x Interfaz.java
Source Design History

308     int i, izq, der = 0;
309
310     for (i = 0; i < 5; i++) {
311
312         izq = i - 1;
313
314         if (izq < 0) {
315             izq = 4;
316         }
317         der = i;
318
319         com = new Filosofo(i, tenedor[izq], tenedor[der], filosofo[i], resultado[i], TextArea_Procesos); //Arreglo del TextArea.
320     }
321 }
322
323
324
325 public static void main(String args[]) {
326     /* Set the Nimbus look and feel */
327     Look and feel setting code (optional)
328
329     /* Create and display the form */
330     java.awt.EventQueue.invokeLater(new Runnable() {
331         public void run() {
332             new Interfaz().setVisible(true);
333         }
334     });
335 }
336
337 // Variables declaration - do not modify
338 private javax.swing.JTextArea TextArea_Procesos;
339 private javax.swing.JButton buttonFilosofo1;
340 private javax.swing.JButton buttonFilosofo2;
341 private javax.swing.JButton buttonFilosofo3;
342 private javax.swing.JButton buttonFilosofo4;
343 private javax.swing.JButton buttonFilosofo5;
344 private javax.swing.JButton jButtonInicio;
345 private javax.swing.JButton jButtonSalir;
346 private javax.swing.JPanel jPanel1;
347 private javax.swing.JPanel jPanelMesa;
348 private javax.swing.JScrollPane jScrollPane1;
349 private javax.swing.JLabel labelFiloi1;
350 private javax.swing.JLabel labelFiloi2;
351 private javax.swing.JLabel labelFiloi3;
352 private javax.swing.JLabel labelFiloi4;
353 private javax.swing.JLabel labelFiloi5;
354 private javax.swing.JLabel labelFiloiCom1;
355 private javax.swing.JLabel labelFiloiCom2;
356 private javax.swing.JLabel labelFiloiCom3;
357 private javax.swing.JLabel labelFiloiCom4;
358 private javax.swing.JLabel labelFiloiCom5;
359 private javax.swing.JLabel labelTenedor1;
360 private javax.swing.JLabel labelTenedor2;
361 private javax.swing.JLabel labelTenedor3;
362 private javax.swing.JLabel labelTenedor4;
363 private javax.swing.JLabel labelTenedor5;
364 private javax.swing.JLabel labelTitulo;
365 // End of variables declaration
366
367
368 class Mesa extends JPanel { //Código para incluir la imagen de la mesa.
369
370     private Image imagen;
371
372     @Override
373     public void paint(Graphics g) {
374         imagen = new ImageIcon(getClass().getResource("/Imagen/mesa.png")).getImage();
375         g.drawImage(imagen, 0, 0, getWidth(), getHeight(), this);
376         setOpaque(false);
377         super.paint(g);
378     }
379 }
380
381
382 }
```

Imagen 6. Captura de pantalla del código formulado en NetBeans, JFrame Form *Interfaz.java*.

El programa en ejecución se puede observar en la **Imagen 7** e **Imagen 8**.

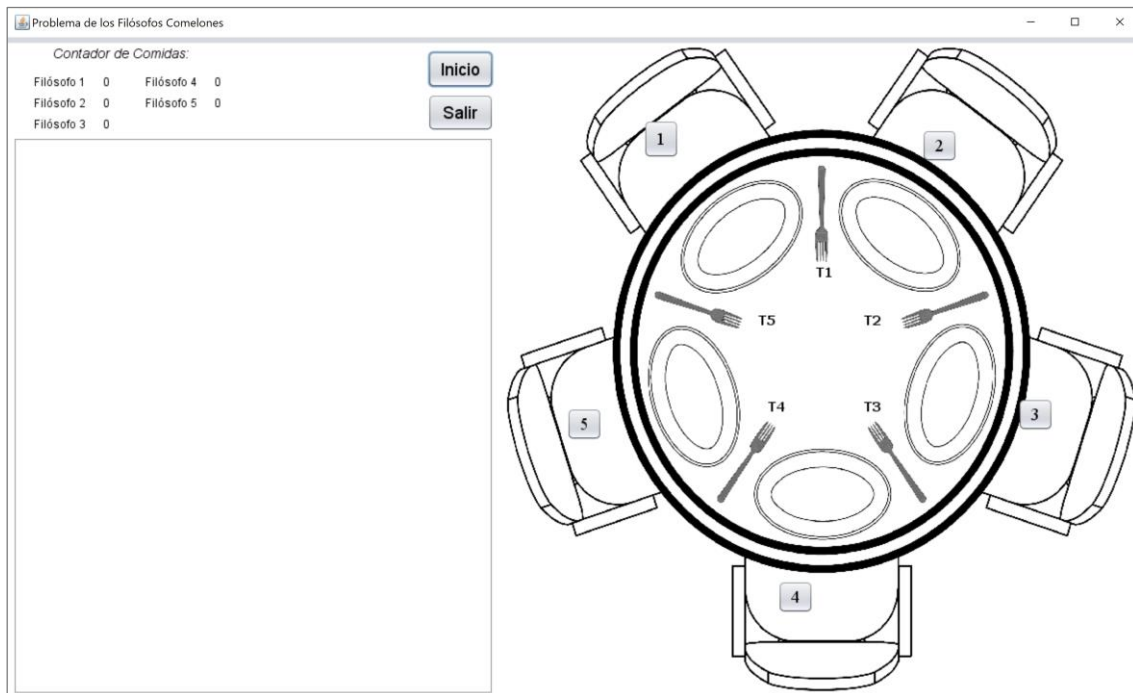


Imagen 7. Captura de pantalla del programa corriendo antes de ser ejecutado.

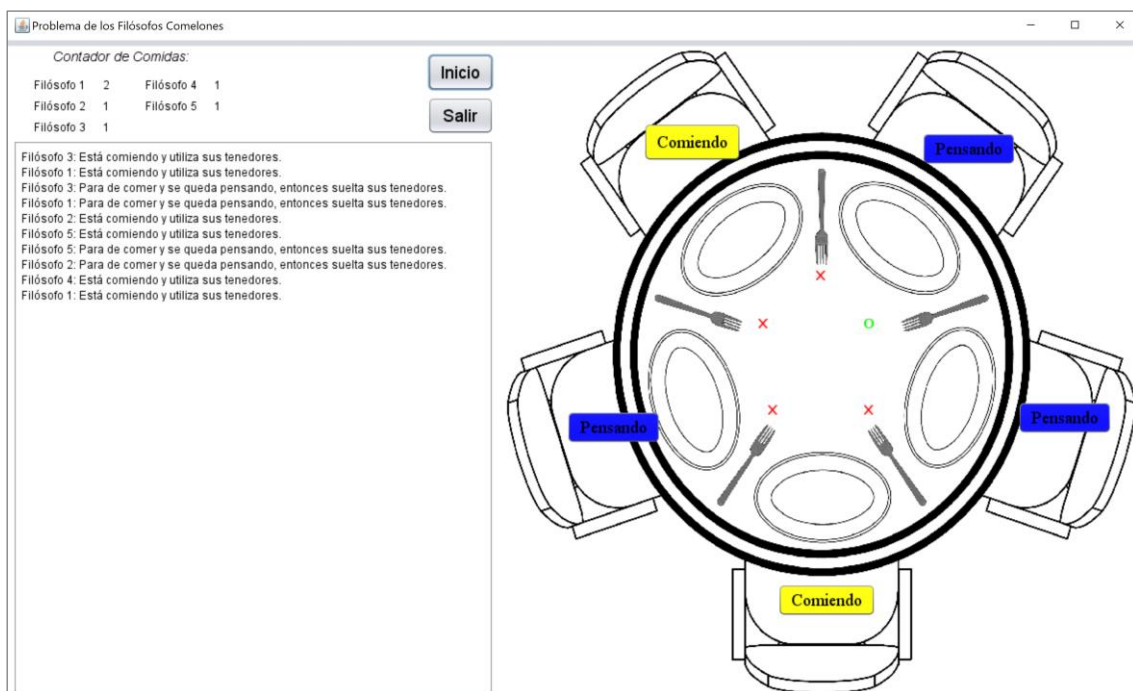
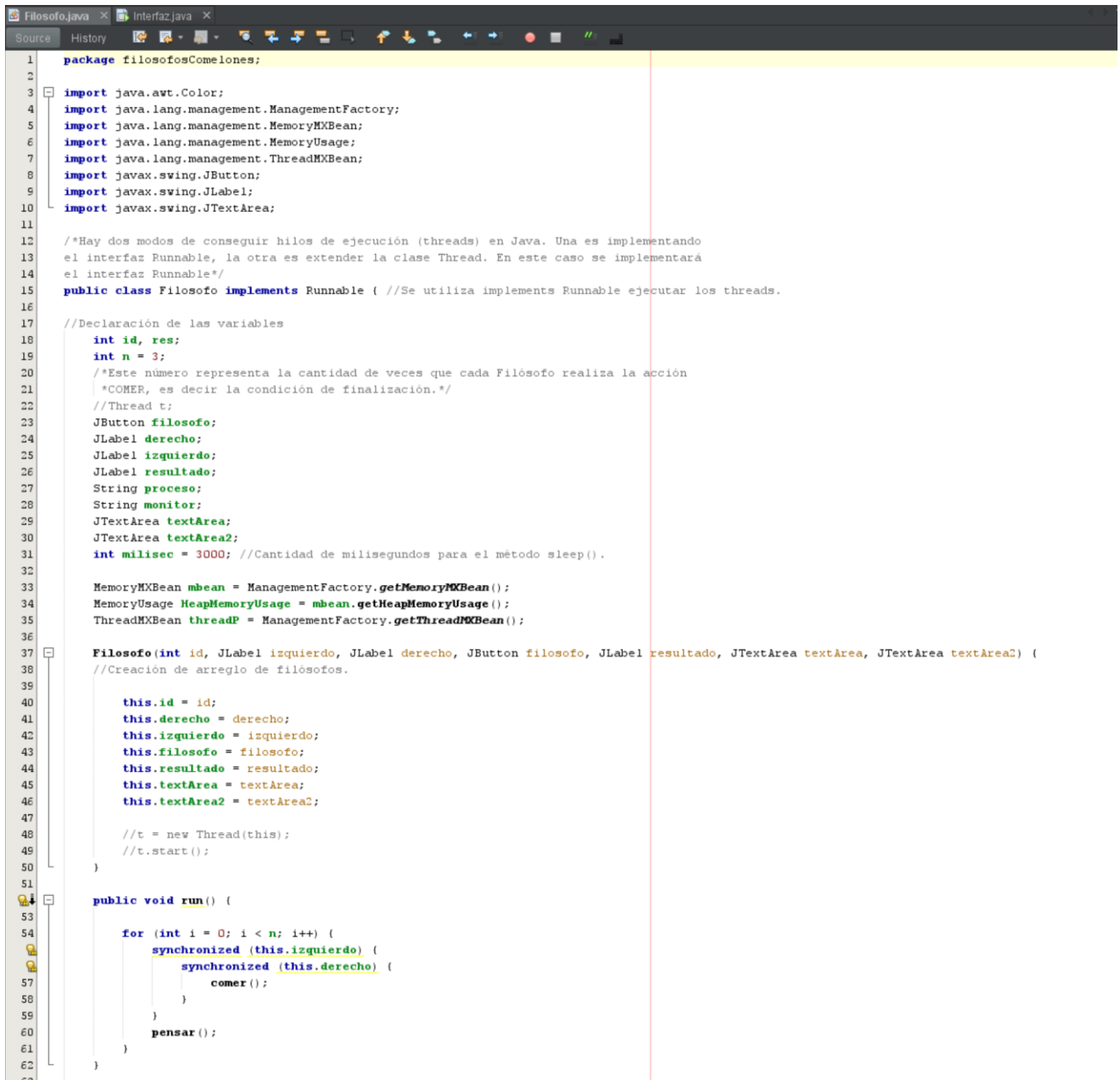


Imagen 8. Captura de pantalla del programa siendo ejecutado.

Posteriormente se determinó que era posible realizar algunas mejoras al programa y además de generar el *Thread* por medio de un pool de *Threads*; para esto se utilizó el siguiente código:



```
1 package filosofosComelones;
2
3 import java.awt.Color;
4 import java.lang.management.ManagementFactory;
5 import java.lang.management.MemoryMXBean;
6 import java.lang.management.MemoryUsage;
7 import java.lang.management.ThreadMXBean;
8 import javax.swing.JButton;
9 import javax.swing.JLabel;
10 import javax.swing.JTextArea;
11
12 /*Hay dos modos de conseguir hilos de ejecución (threads) en Java. Una es implementando
13 el interfaz Runnable, la otra es extender la clase Thread. En este caso se implementará
14 el interfaz Runnable*/
15 public class Filosofo implements Runnable { //Se utiliza implements Runnable ejecutar los threads.
16
17 //Declaración de las variables
18     int id, res;
19     int n = 3;
20     /*Este número representa la cantidad de veces que cada Filósofo realiza la acción
21     *COMER, es decir la condición de finalización.*/
22     //Thread t;
23     JButton filosofo;
24     JLabel derecho;
25     JLabel izquierdo;
26     JLabel resultado;
27     String proceso;
28     String monitor;
29     JTextArea textArea;
30     JTextArea textArea2;
31     int milisec = 3000; //Cantidad de milisegundos para el método sleep().
32
33     MemoryMXBean mbean = ManagementFactory.getMemoryMXBean();
34     MemoryUsage heapMemoryUsage = mbean.getHeapMemoryUsage();
35     ThreadMXBean threadP = ManagementFactory.getThreadMXBean();
36
37     Filosofo(int id, JLabel izquierdo, JLabel derecho, JButton filosofo, JLabel resultado, JTextArea textArea, JTextArea textArea2) {
38         //Creación de arreglo de filósofos.
39
40         this.id = id;
41         this.derecho = derecho;
42         this.izquierdo = izquierdo;
43         this.filosofo = filosofo;
44         this.resultado = resultado;
45         this.textArea = textArea;
46         this.textArea2 = textArea2;
47
48         //t = new Thread(this);
49         //t.start();
50     }
51
52     public void run() {
53
54         for (int i = 0; i < n; i++) {
55             synchronized (this.izquierdo) {
56                 synchronized (this.derecho) {
57                     comer();
58                 }
59             }
60             pensar();
61         }
62     }
63 }
```

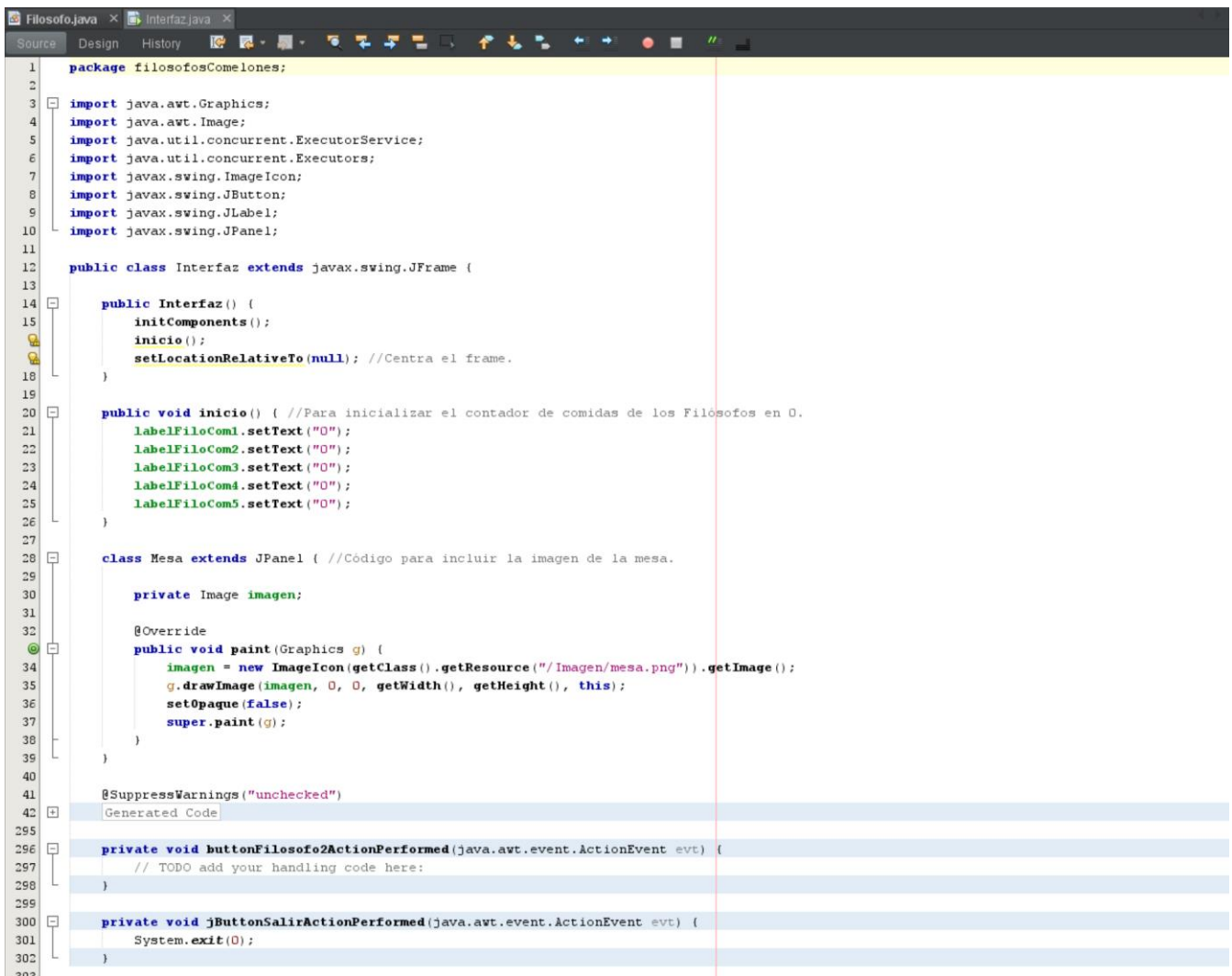
Imagen 9. Captura de pantalla del código formulado en NetBeans, clase *Filosofo.java* con los cambios efectuados.

```

63
64 void comer() { //Funcionalidad Comer.
65     derecho.setText("X"); //Tenedor ocupado.
66     derecho.setForeground(Color.red);
67
68     izquierdo.setText("X"); //Tenedor ocupado.
69     izquierdo.setForeground(Color.red);
70
71     filosofo.setText("Comiendo"); //Acción del Filósofo.
72     filosofo.setBackground(Color.yellow);
73
74     res = Integer.parseInt(resultado.getText());
75     res += 1;
76     resultado.setText(String.valueOf(res));
77
78     proceso = "Filósofo " + (id + 1) + ":" + " Está comiendo y utiliza sus tenedores.\n \n";
79     textArea.append(proceso);
80
81     monitor = "Filósofo " + (id + 1) + " - Thread: " + threadP + ".\n"
82             + "Filósofo " + (id + 1) + " - HeapMemory: " + mbean + ".\n"
83             + "Filósofo " + (id + 1) + " - Acción de Comer (Bytes en memoria para la asignación de objetos):\n" + HeapMemoryUsage + ".\n \n";
84     textArea2.append(monitor);
85
86     try {
87         Thread.sleep(milisec); //El método sleep() le indica al thread que "duerma" durante los milisegundos especificados.
88     } catch (InterruptedException e) {
89     }
90
91     derecho.setText("O"); //Tenedor Libre.
92     derecho.setForeground(Color.green);
93
94     izquierdo.setText("O"); //Tenedor Libre.
95     izquierdo.setForeground(Color.green);
96
97     filosofo.setText("Pensando"); //Acción del Filósofo.
98     filosofo.setBackground(Color.blue);
99
100    proceso = "Filósofo " + (id + 1) + ":" + " Para de comer y se queda pensando, entonces suelta sus tenedores.\n \n";
101    textArea.append(proceso);
102
103    monitor = "Filósofo " + (id + 1) + " - Thread: " + threadP + ".\n"
104            + "Filósofo " + (id + 1) + " - HeapMemory: " + mbean + ".\n"
105            + "Filósofo " + (id + 1) + " - Acción de Pensar (Bytes en memoria para la asignación de objetos):\n" + HeapMemoryUsage + ".\n \n";
106    textArea2.append(monitor);
107
108
109 void pensar() { //Funcionalidad Pensar.
110     derecho.setText("O"); //Tenedor libre.
111     derecho.setForeground(Color.green);
112
113     izquierdo.setText("O"); //Tenedor libre.
114     izquierdo.setForeground(Color.green);
115
116     filosofo.setText("Pensando"); //Acción del Filósofo.
117     filosofo.setBackground(Color.blue);
118
119     try {
120         Thread.sleep(milisec); //El método sleep() le indica al thread que "duerma" durante los milisegundos especificados.
121     } catch (InterruptedException e) {
122     }
123
124 }

```

Imagen 10. Captura de pantalla del código formulado en NetBeans, clase *Filosofo.java* con los cambios efectuados.



```
1 package filosofosComelones;  
2  
3 import java.awt.Graphics;  
4 import java.awt.Image;  
5 import java.util.concurrent.ExecutorService;  
6 import java.util.concurrent.Executors;  
7 import javax.swing.ImageIcon;  
8 import javax.swing.JButton;  
9 import javax.swing.JLabel;  
10 import javax.swing.JPanel;  
11  
12 public class Interfaz extends javax.swing.JFrame {  
13  
14     public Interfaz() {  
15         initComponents();  
16         inicio();  
17         setLocationRelativeTo(null); //Centra el frame.  
18     }  
19  
20     public void inicio() { //Para inicializar el contador de comidas de los Filósofos en 0.  
21         labelFiloCom1.setText("0");  
22         labelFiloCom2.setText("0");  
23         labelFiloCom3.setText("0");  
24         labelFiloCom4.setText("0");  
25         labelFiloCom5.setText("0");  
26     }  
27  
28     class Mesa extends JPanel { //Codigo para incluir la imagen de la mesa.  
29  
30         private Image imagen;  
31  
32         @Override  
33         public void paint(Graphics g) {  
34             imagen = new ImageIcon(getClass().getResource("/Imagen/mesa.png")).getImage();  
35             g.drawImage(imagen, 0, 0, getWidth(), getHeight(), this);  
36             setOpaque(false);  
37             super.paint(g);  
38         }  
39     }  
40  
41     @SuppressWarnings("unchecked")  
42     Generated Code  
295  
296     private void buttonFilosofo2ActionPerformed(java.awt.event.ActionEvent evt) {  
297         // TODO add your handling code here:  
298     }  
299  
300     private void jButtonSalirActionPerformed(java.awt.event.ActionEvent evt) {  
301         System.exit(0);  
302     }  
303 }
```

Imagen 11. Captura de pantalla del código formulado en NetBeans, JFrame Form *Interfaz.java* con los cambios efectuados.


```

303
304 //Código para dar funcionalidad al botón de inicio.
305 private void jButtonInicioActionPerformed(java.awt.event.ActionEvent evt) {
306
307     int fil = 5; //Cantidad de Filósofos.
308     int i = 0;
309     int izq = 0;
310     int der = 0;
311     ExecutorService threadPool = Executors.newCachedThreadPool();
312
313     //Filosofo com; //Declaracion de la variable filósofo.
314     //Pruebas para la creación de un Thread con el método Thread.
315
316     JButton filosofo[];
317     filosofo = new JButton[5];
318     filosofo[0] = buttonFilosofo1;
319     filosofo[1] = buttonFilosofo2;
320     filosofo[2] = buttonFilosofo3;
321     filosofo[3] = buttonFilosofo4;
322     filosofo[4] = buttonFilosofo5;
323
324     JLabel tenedor[];
325     tenedor = new JLabel[5];
326     tenedor[0] = labelTenedor1;
327     tenedor[1] = labelTenedor2;
328     tenedor[2] = labelTenedor3;
329     tenedor[3] = labelTenedor4;
330     tenedor[4] = labelTenedor5;
331
332     JLabel resultado[];
333     resultado = new JLabel[5];
334     resultado[0] = labelFileCom1;
335     resultado[1] = labelFileCom2;
336     resultado[2] = labelFileCom3;
337     resultado[3] = labelFileCom4;
338     resultado[4] = labelFileCom5;
339
340     for (i = 0; i < fil; i++) {
341         izq = i - 1;
342         if (izq < 0) {
343             izq = (fil - 1);
344             //izq = 4;
345             //Pruebas para la creación de un Thread con el método Thread.
346         }
347         der = i;
348
349         threadPool.execute(new Filosofo(i, tenedor[izq], tenedor[der], filosofo[i], resultado[i], TextArea_Procesos, TextArea_Monitor));
350         //com = new Filosofo(i, tenedor[izq], tenedor[der], filosofo[i], resultado[i], TextArea_Procesos, TextArea_Monitor); //Instancia de
351         //Pruebas para la creación de un Thread con el método Thread.
352     }
353     threadPool.shutdown();
354 }
355
356 public static void main(String args[]) {
357     /* Set the Nimbus look and feel */
358
359     Look and feel setting code (optional)
360     /* Create and display the form */
361     java.awt.EventQueue.invokeLater(new Runnable() {
362         public void run() {
363             new Interfaz().setVisible(true);
364         }
365     });
366
367     // Variables declaration - do not modify
368     private javax.swing.JTextArea TextArea_Monitor;
369     private javax.swing.JTextArea TextArea_Procesos;
370     private javax.swing.JButton buttonFilosofo1;
371     private javax.swing.JButton buttonFilosofo2;
372     private javax.swing.JButton buttonFilosofo3;
373     private javax.swing.JButton buttonFilosofo4;
374     private javax.swing.JButton buttonFilosofo5;
375     private javax.swing.JButton jButtonInicio;
376     private javax.swing.JButton jButtonSalir;
377     private javax.swing.JPanel jPanel1;
378     private javax.swing.JPanel jPanelMesa;
379     private javax.swing.JScrollPane jScrollPane1;
380     private javax.swing.JScrollPane jScrollPane2;
381     private javax.swing.JLabel labelFile1;
382     private javax.swing.JLabel labelFile2;
383     private javax.swing.JLabel labelFile3;
384     private javax.swing.JLabel labelFile4;
385     private javax.swing.JLabel labelFile5;
386     private javax.swing.JLabel labelFileCom1;
387     private javax.swing.JLabel labelFileCom2;
388     private javax.swing.JLabel labelFileCom3;
389     private javax.swing.JLabel labelFileCom4;
390     private javax.swing.JLabel labelFileCom5;
391     private javax.swing.JLabel labelTenedor1;
392     private javax.swing.JLabel labelTenedor2;
393     private javax.swing.JLabel labelTenedor3;
394     private javax.swing.JLabel labelTenedor4;
395     private javax.swing.JLabel labelTenedor5;
396     private javax.swing.JLabel labelTitulo;
397     private javax.swing.JLabel labelTitulo2;
398     // End of variables declaration
399 }

```

Imagen 12. Captura de pantalla del código formulado en NetBeans, JFrame Form *Interfaz.java* con los cambios efectuados.

En la **Imagen 9** e **Imagen 10** se muestran los cambios realizados en la clase *Filosofo.java* y en la **Imagen 11** e **Imagen 12** los cambios del JFrame Form *Interfaz.java*. En la clase *Filosofo.java* se incluyeron algunos elementos para poder conocer el comportamiento de los **Threads** mientras se ejecuta el programa, esto se realizó por medio del **ManagementFactory**, el **MemoryUsage** y **ThreadMXBean**. Además, se quitó de la clase *Filosofo.java* el objeto de tipo *Thread t*. Luego en el JFrame Form *Interfaz.java*, se generó un pool de **threads** por medio del **ExecutorService**.

El programa en ejecución con los cambios realizados se puede apreciar en la **Imagen 13**.

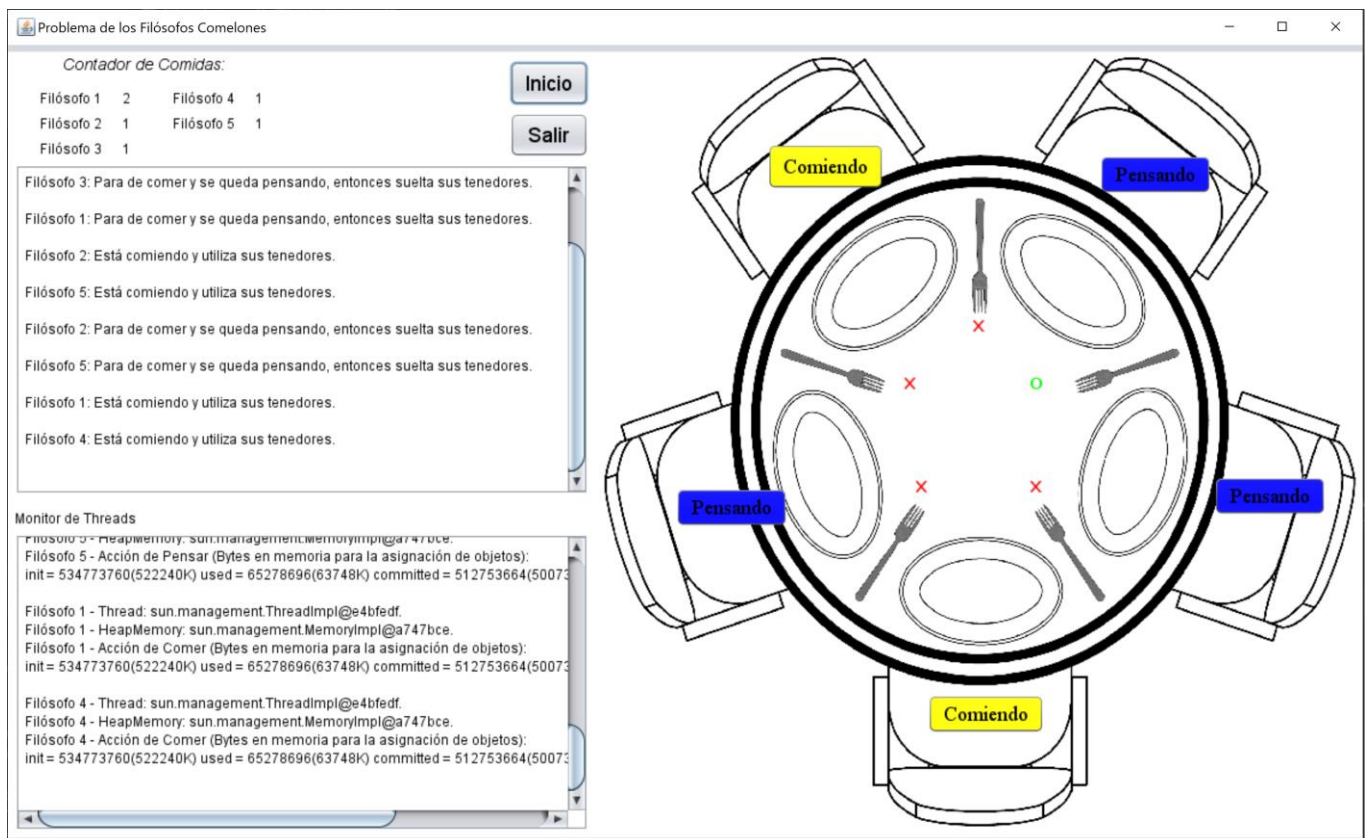


Imagen 13. Captura de pantalla del programa con los cambios realizados en el código en ejecución.

Microservicios y una Implementación Conceptual al Problema de los Filósofos Comelones

Antes de poder definir en que consiste la **arquitectura de microservicios** es necesario definir lo que es la **arquitectura monolítica**; ya que son conceptos que se encuentran relacionados.

¿Qué es una Arquitectura Monolítica?

Una **arquitectura monolítica** es un modelo tradicional de un programa de software que se compila como una unidad unificada y que es autónoma e independiente de otras aplicaciones. Además, la **arquitectura monolítica** es una red informática grande y única,

con una base de código que incluye todos elementos que se encuentran en una aplicación. Por esta razón para hacer cambios en este tipo de aplicaciones, hay que actualizar toda la pila, lo que requiere acceder a la base de código y compilar e implementar una versión actualizada de la interfaz del lado del servicio. Esto hace que las actualizaciones sean restrictivas y lentas. Los monolitos pueden resultar prácticos al principio de un proyecto para aliviar la sobrecarga cognitiva de la gestión de código, así como la implementación. De esta forma, es posible publicar a la vez todo lo presente en el monolito.

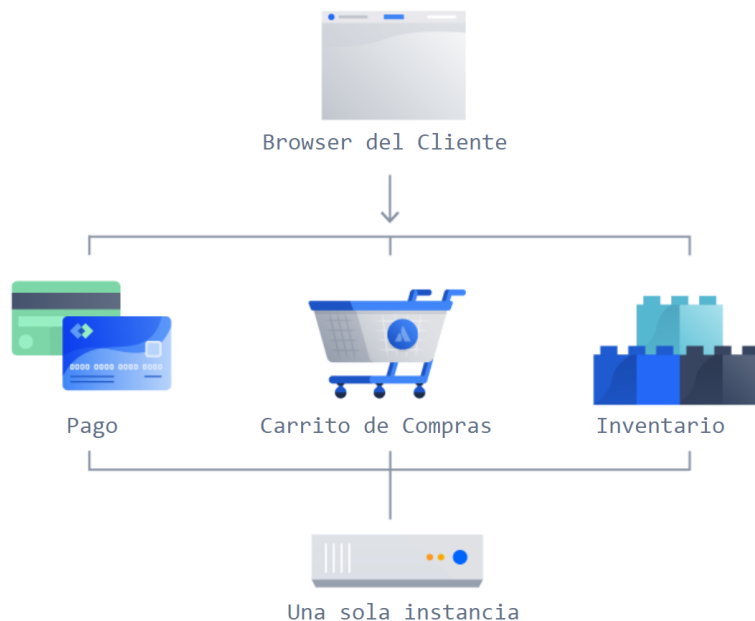


Imagen 14. Ejemplo gráfico de una *Arquitectura Monolítica*.

¿Qué son los Microservicios?

Una *arquitectura de microservicios*, o simplemente *microservicios*, es un método de arquitectura que se basa en una serie de servicios que se pueden implementar de forma independiente. Estos servicios tienen su propia lógica dentro de la aplicación y su propia base de datos con un objetivo específico. La actualización, las pruebas, la implementación y el escalado se llevan a cabo dentro de cada servicio. Los *microservicios* desacoplan los principales intereses específicos de dominios de la aplicación en bases de código independientes. Los *microservicios* no reducen la complejidad, pero hacen que cualquier complejidad sea visible y más gestionable, ya que separan las tareas en procesos más pequeños que funcionan de manera independiente entre sí y contribuyen al conjunto global. La adopción de *microservicios* suele ir de la mano de *DevOps*²², ya que son la

²² El término *DevOps*, es una combinación de los términos ingleses *Development* (desarrollo) y *Operations* (operaciones), con este se designa la unión de personas, procesos y tecnología para ofrecer valor a los clientes de forma constante. *DevOps* es un marco de trabajo y una filosofía en constante evolución que promueve un mejor desarrollo de aplicaciones en menos tiempo y la rápida publicación de nuevas o revisadas funciones de software o productos para los clientes. Con *DevOps* se promueve una comunicación continua más fluida, la colaboración, la integración, la visibilidad y la transparencia entre equipos de desarrollo de aplicaciones (*Dev*) y sus homólogos en operaciones tecnológicas (*Ops*).

base de las prácticas de entrega continua con las que los equipos pueden adaptarse rápidamente a los requisitos de los usuarios.

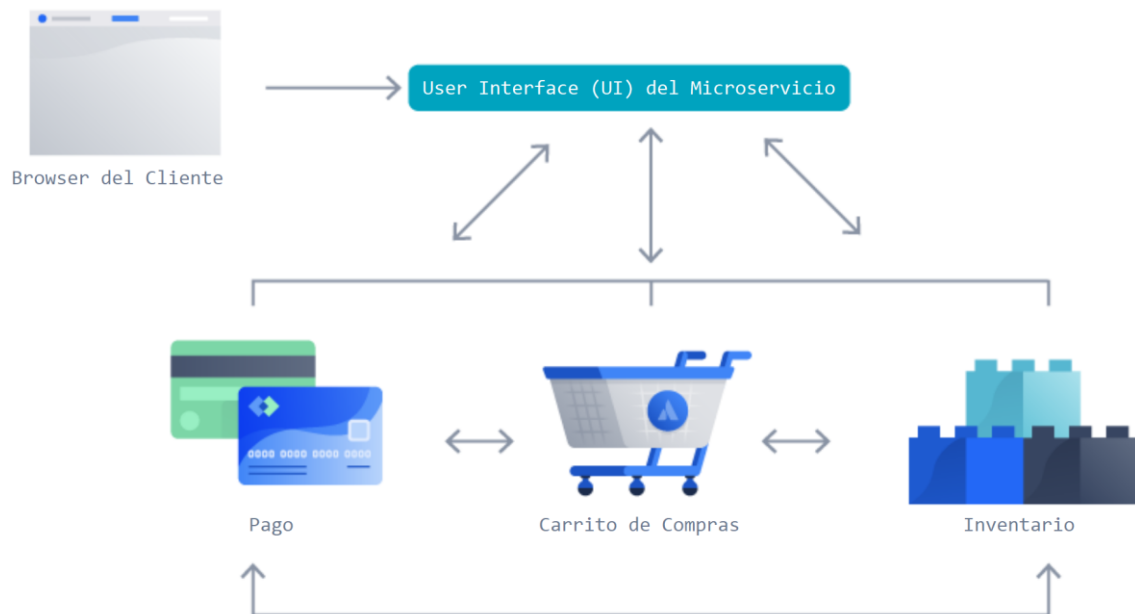


Imagen 15. Ejemplo gráfico de una *Arquitectura de Microservicios*.

En el artículo escrito por James Lewis y Martin Fowler, “*Microservices, a definition of this new architectural term*”, definen a los microservicios como;

*“En pocas palabras, el estilo arquitectónico de microservicios es un enfoque para el desarrollo de una sola aplicación como un conjunto de servicios pequeños, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos ligeros, (...).”*²³

Otra definición de **microservicios** se presenta en el documento “*Guía de Microservicios Punto de vista*” publicado por IBM la cual es;

*“(…) En el estilo arquitectónico de microservicios, los desarrolladores construyen y empaquetan de manera independiente varias aplicaciones más pequeñas y a su vez cada una implementa partes de la aplicación (...).”*²⁴

Adicionalmente en el precitado documento se mencionan las reglas básicas que controlan la implementación de aplicaciones utilizando la **arquitectura de microservicios**, estas son:

²³ Texto citado del artículo “*Microservices, a definition of this new architectural term*” – 25 de marzo de 2014, escrito por James Lewis y Martin Fowler; extraído del sitio web <https://martinfowler.com/articles/microservices.html>

²⁴ Extraído del sitio web <https://www.ibm.com/downloads/cas/5O8YOPKN>.

1. ***Dividir un sistema monolítico grande en muchos servicios pequeños*** – Un solo servicio accesible por red es la unidad más pequeña utilizada para una aplicación de microservicios. Cada servicio ejecuta su propio proceso. Esta regla se denomina un servicio por contenedor. Contenedor se refiere a un contenedor Docker o cualquier otro mecanismo de despliegue ligero tales como un tiempo de ejecución de Cloud Foundry.
2. ***Optimizar servicios para una sola función*** – En un enfoque SOA monolítico tradicional, una aplicación en tiempo de ejecución actúa sobre múltiples funciones del negocio. En un enfoque de microservicios, hay solamente una función comercial por servicio. Esto hace a cada servicio más pequeño y más simple de escribir y mantener. Esto se conoce como el Principio de Responsabilidad Única (SRP).
3. ***Comunicarse a través de REST API y message brokers*** – Una de las desventajas del enfoque SOA es que hay numerosos estándares y opciones para implementar los servicios SOA. El enfoque de microservicios limita estrictamente los tipos de conectividad de red que un servicio puede implementar para lograr máxima simplicidad. Del mismo modo, los microservicios tienden a evitar el fuerte acoplamiento introducido por la comunicación implícita a través de una base de datos. Toda comunicación de servicio a servicio debe ser a través del servicio API o al menos debe usar un patrón de comunicación explícito tal como el patrón Claim Check [Hohpe y Woolf].
4. ***Aplicar CI/CD²⁵ por servicio*** – En una aplicación grande compuesta de muchos servicios, diferentes servicios evolucionan a ritmos diferentes. Cada servicio tiene una interconexión continua única de integración/entrega que permite que proceda a un paso natural. Esto no es posible con el enfoque monolítico, donde cada aspecto del sistema es liberado a la fuerza a la velocidad del componente más lento del sistema.
5. ***Aplicar alta disponibilidad por servicio (HA)/decisiones sobre clústeres*** – Cuando se construyen sistemas grandes, la agrupación en clústeres no es un enfoque del tipo en un tamaño entra todo. El enfoque monolítico de escalar todos los servicios al mismo nivel lleva a la sobrecarga de algunos servidores y al desperdicio de otros; o aún peor, la inanición de algunos servicios por otros que monopolizan todos los recursos compartidos disponibles, tales como agrupaciones de subprocesos. La realidad es que, en un sistema grande, no se necesita escalar todos los servicios; muchos servicios pueden ser desplegados en un número mínimo de servidores para conservar recursos. Otros requieren ampliarse a números muy grandes.

²⁵ Se refiere a las prácticas combinadas de integración continua (Continuous Integration, CI) y entrega continua (Continuous Delivery, CD)

Además, en ese mismo documento, se menciona que una “(...) de las diferencias clave entre microservicios y paradigmas como SOA²⁶ y API es el foco en los componentes desplegados y en ejecución. Los microservicios se enfocan en el nivel de detalle de los componentes desplegados más que en las interfaces (...)”.

Básicamente y de una manera resumida es posible concluir que una “(...) aplicación monolítica se compila como una sola unidad unificada, mientras que una arquitectura de microservicios es una serie de servicios pequeños que se pueden implementar de forma independiente (...)”²⁷; tomando en consideración los conceptos antes mencionados y reglas básicas para la implementación de los **microservicios**, es posible realizar la siguiente analogía entre el problema de los filósofos y la aplicación de la arquitectura de microservicios.

- Considerando que los **microservicios** buscan dividir un *sistema monolítico*²⁸ grande en muchos servicios pequeños es posible pensar en un símil de los elementos del **problema de los filósofos comelones** y los **microservicios**; de esta manera el sistema (aplicación) puede considerarse como la mesa, los servicios como los filósofos, los recursos computacionales como los tenedores y la función propia de cada servicio como las acciones de los filósofos (comer/pensar); ver **Imagen 16**.

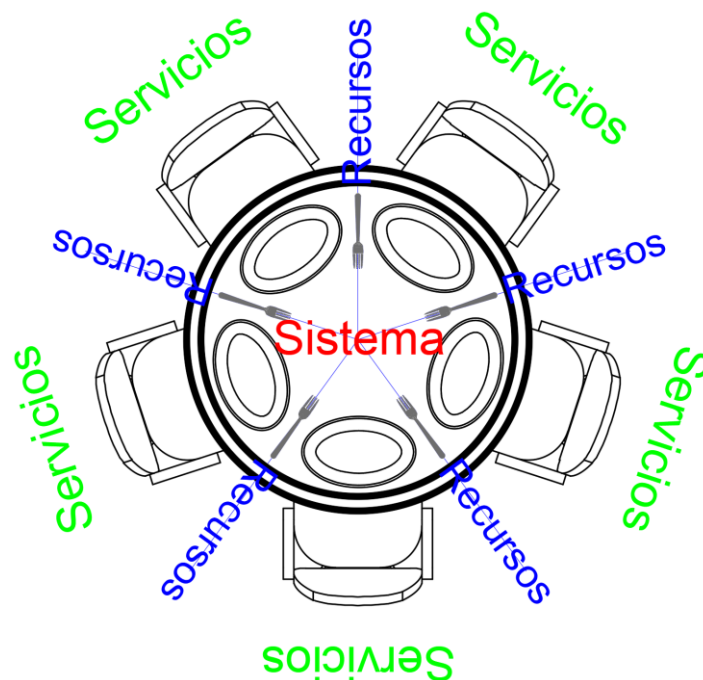


Imagen 16. Representación gráfica de la analogía entre los microservicios y el problema de los filósofos.

²⁶ SOA, Service-Oriented Architecture – en español Arquitectura Orientada a Servicios.

²⁷ Extraído del sitio web <https://www.atlassian.com/es/microservices/microservices-architecture/microservices-vs-monolith>.

²⁸ Los sistemas monolíticos (también definidos como aplicaciones monolíticas) agrupan la funcionalidad y sus servicios en una base de código única.

Así, es posible confirmar la analogía anterior; ya que una aplicación (*mesa*) posee una memoria global en la que se comparten los recursos (*tenedores*), esta misma aplicación (*mesa*) requiere de varios **microservicios** (*filósofos que realizan la acción de comer o pensar, según los parámetros establecidos de tener o no los tenedores*) que se utilizan los recursos (*tenedores*); además por medio de los sistemas de archivos es posible validar si dichos recursos (*tenedores*) están disponibles (similar al problema de los filósofos que únicamente pueden comer si poseen ambos tenedores) para poder completar el **microservicio**.

En el sitio web <https://github.com/OAPV27/FilosofosComelones> es posible acceder al código generado y documentación elaborado.

Bibliografía

Tanenbaum, A. (México, 2009). *Sistemas operativos modernos (3ra ed.)*. Pearson: Prentice Hall.

Diego Toapanta. (11 de julio de 2020). *Solución Filósofos Comensales Java Hilos Sincronización* [Video]. YouTube. Sitio web consultado <https://www.youtube.com/watch?v=naW6FFbwSdM&t=953s>, el 25 de julio de 2022.

Toapanta, D. (n.d.). *ComenzalesPFrar*. Google Docs. Código consultado del sitio web <https://drive.google.com/file/d/1hip4ItyojAtomPiFNeeC2yttVgUgsoFa/view>, el 25 de julio de 2022.

Portillo, P. (1 de febrero de 2020). *La Cena de los Filósofos*. Paco Portillo - Programador Senior Java. Sitio web consultado <https://pacoportillo.es/informatica-avanzada/programacion-multiproceso/la-cena-de-los-filosofos/>, el 25 de julio de 2022.

Tutorial de Java - Creación y Control de Hilos. (n.d.). Tutorial de Java Tecnology. Sitio web consultado <http://dis.um.es/%7Ebmmoros/Tutorial/parte10/cap10-2.html>, el 6 de agosto de 2022.

Thread (Java Platform SE 7). (24 de junio de 2020). Oracle. Sitio web consultado <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>, el 7 de agosto de 2022.

Rodríguez, D. D. (n.d.). *Introducción a la Concurrency en Java (I)*. Blog Softtek. Sitio web <https://blog.softtek.com/es/java-concurrency>, el 10 de agosto de 2022.

Dining Philosophers Problem. (23 de octubre de 2021). InterviewBit. Sitio web consultado <https://www.interviewbit.com/blog/dining-philosophers-problem/>, el 11 de agosto de 2022.

Mayorov, A. (14 de marzo de 2021). *The Dining Philosophers problem and different ways of solving it.* ZeroBone. Sitio web consultado <https://zerobone.net/blog/cs/dining-philosophers-problem/>, el 11 de agosto de 2022.

Problema de la cena de los filósofos en Java. (12 de junio de 2022). Disco Duro de Roer. Sitio web consultado <https://www.discoduroderoer.es/problema-de-la-cena-de-los-filosofos-en-java/>, el 11 de agosto de 2022.

Lewis, J., & Fowler, M. (n.d.). Microservices. MartinFowler.Com. Sitio web consultado <https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture>, el 12 de agosto de 2022.

De Monolítico a Microservicios - Dharwin Aarón Pérez Rodríguez - PyCon Latam 2019. (10 de octubre de 2019). [Video]. YouTube. Sitio web consultado <https://www.youtube.com/watch?v=2abOxb49aBU>, el 14 de agosto de 2022.

Atlassian. (n.d.). *Comparación entre la arquitectura monolítica y la arquitectura de microservicios.* Sitio web consultado <https://www.atlassian.com/es/microservices/microservices-architecture/microservices-vs-monolith>, el 16 de agosto de 2022.

Fowler, M. (21 de agosto de 2019). *Microservices Guide.* MartinFowler.Com. Sitio web consultado <https://martinfowler.com/microservices/>, el 16 de agosto de 2022.

C. (7 de octubre de 2021). Comunicación entre microservicios: Métodos, tipos y estilos. Chakray. Sitio web consultado, <https://www.chakray.com/es/comunicacion-entre-microservicios-metodos-tipos-y-estilos/>, el 22 de agosto de 2022.

López, J. (18 de marzo de 2019). *Midiendo el uso real de memoria: JMnemohistosyne.* Spartan Blog. Sitio web consultado, <https://www.jerolba.com/midiendo-el-uso-real-de-memoria-jmnemohistosyne/>, el 22 de agosto de 2022.