**“Reporte del análisis sobre el caso de estudio #1”**

Nicolas Perez Teran - 202116903,

Daniel Andrés Bernal Cáceres - 202020706,

Isaac David Bermúdez - 202014146.

Documentación y explicación al caso de estudio.

Universidad de los Andes, Bogotá, Colombia

[n.perezt2@uniandes.edu.co](mailto:n.perezt2@uniandes.edu.co)  -  [da.bernalc1@uniandes.edu.co](mailto:da.bernalc1@uniandes.edu.co)

[i.bermudezl@uniandes.edu.co](mailto:i.bermudezl@uniandes.edu.co)

Fecha de presentación: febrero 23 de 2023. 

**Tabla de contenido**

1. Introducción. 3

2. Descripción del caso. 3

2.1. Restricciones. 3

2.2 Actores involucrados en la solución. 4

3. Descripción de la implementación. 4

3.1. Diagrama de Clases: 4

3.2. Productos: 5

3.3. Buzones: 5

3.4. Trabajadores: 5

3.5. Asignador de identificador: 5

4.1. Implementación de trabajadores. 5

4.1.1. OrangeWorker y BlueWorker. 6

4.1.2. RedWorker. 6

4.2. Implementación de producto. 7

4.2.1. GenerateId. 7

4.3. PRODUCT FACTORY. 8

4.3.1 Product factory. 8

5. Validación 9

6. Glosario. 9

6.1 Main. 9

6.2 ProductFactoryBuilder. 9

6.3 ProductFactory. 10

6.4 Workers. 11

6.4.1 BlueWorker. 11

6.4.2 OrangeWorker. 12

6.4.3 RedWorker. 13

6.5 Mailbox. 14

6.5.1 FiniteMailbox. 14

6.5.2 InfiniteMailbox. 15

6.6 Product. 15

6.7 GenerateId. 16

# Introducción.

El objetivo de este documento se centra en la explicación detallada del diseño e implementación de la solución correspondiente al caso de estudio número uno, el cual está enfocado en el uso de concurrencia para modelar un escenario determinado. Este informe lo realizaremos mediante la explicación de los actores implicados, objetos implementados, especificación de cada etapa, argumentación de los canales de comunicación y justificaciones referentes al uso de concurrencia utilizado.

# Descripción del caso.

Para este primer caso, se propone un escenario alusivo al funcionamiento de una fábrica de dos productos: productos azules y naranjas. En particular, esta fábrica funciona mediante tres fases donde se realiza una acción con cada uno de los productos; en la primera, habrán trabajadores que se encargan de la creación del producto y lo mandan a la siguiente fase de transformación; en la segunda fase, se hace una transformación (procesamiento) al producto y se envía a la fase tres; en la tercera fase, el producto es finalmente terminado y se direcciona a su última etapa donde se mostrara cada uno de los productos creados durante todo el proceso. Las transiciones entre etapas estarán manejadas por los trabajadores azules y naranjas (cada uno crea y transporta productos de su mismo color) y el trabajador rojo será el encargado de tomar los productos, tanto azules como naranjas, e informar los productos en su estado final.

# 2.1. Restricciones.

Dado el caso anterior, debemos saber que hay restricciones que debe seguir la fábrica. Específicamente, se pide que el número de productos generados por el proceso sea definido al inicio; además, es un comportamiento que se repite entre la capacidad de los medios de transporte de los productos entre etapas. Por último, debemos tener en cuenta que lo trabajadores son limitados, el número de trabajadores por cada etapa se define al inicio y solo uno será de color naranja, es decir, el número de trabajadores en cada etapa es el mismo.

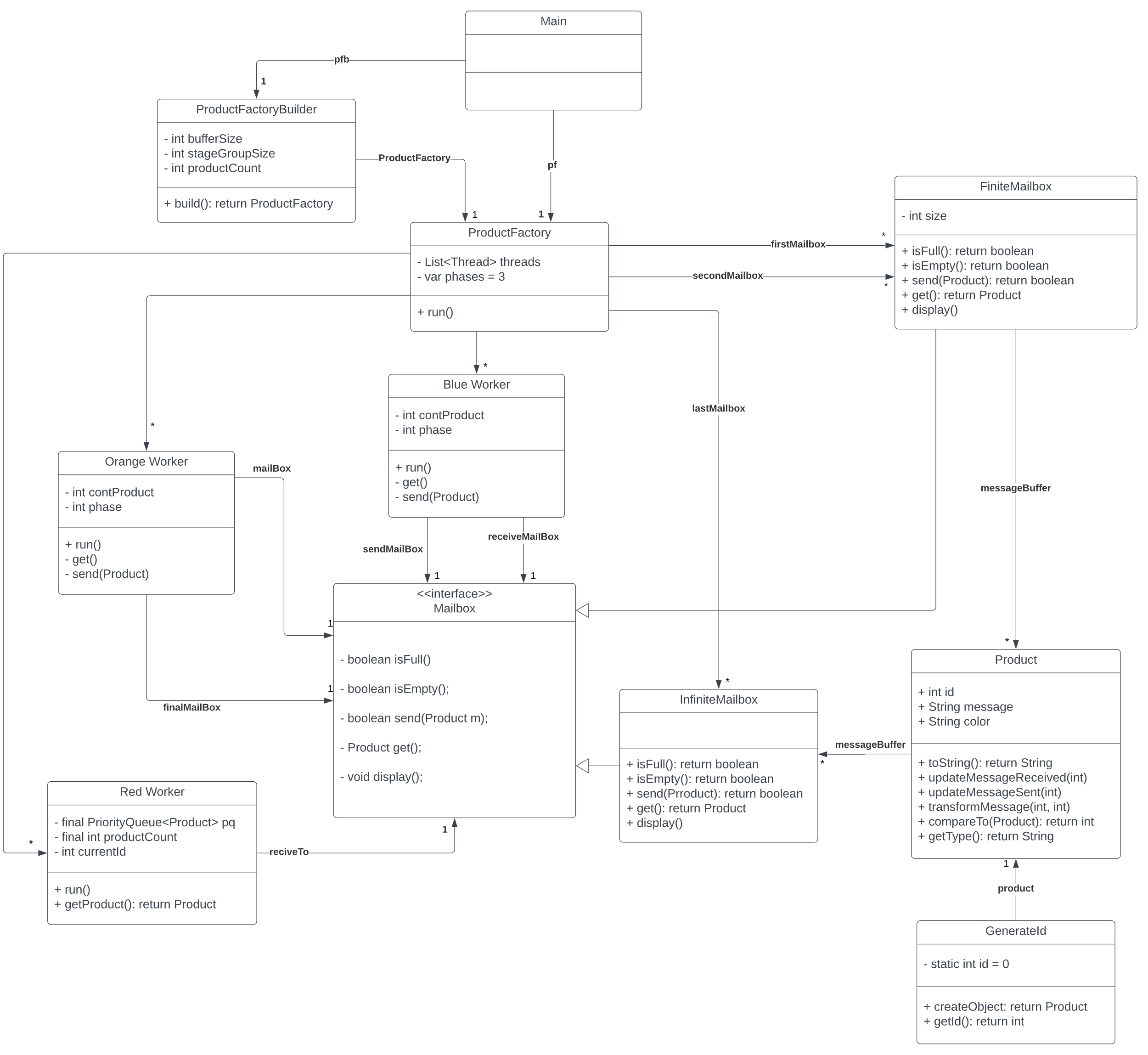
# Actores involucrados en la solución.

Para la solución, se realizó un análisis donde se encontraron que se deben implementar productos, trabajadores, buzones y asignador de identificador para los objetos. Las funciones de cada uno de los elementos mencionados son los siguientes.

# Descripción de la implementación.

Retomando el tema importante, debemos explicar la solución propuesta para este caso.

# 3.1. Diagrama de Clases:



# 3.2. Productos:

Como bien sabemos, los productos son los objetos que la fábrica crea para su distribución. Estos serán creados y transportados a través de etapas donde irán cambiando su estado en un proceso que puede durar entre 50-500 milisegundos.

# 3.3. Buzones:

A lo largo de los procesos podemos evidenciar que habrá transporte de productos entre fases. Sin embargo, sabemos que las fases ocurren al mismo tiempo todas y de manera independiente, lo cual crea la necesidad de tener un medio de transporte del cual extraer o mandar productos; y es con lo anterior que se crean los buzones como medio de comunicación entre etapas. Es importante recordar que entre etapas estos buzones son finitos y su tamaño está determinado en la construcción de la fábrica, aun así, el buzón al final tiene tamaño variable pues será el lugar donde llegará todo lo que se produce.

# 3.4. Trabajadores:

Por otro lado, como parte primordial de la fábrica debemos tomar en cuenta los actores encargados del funcionamiento general. De esta manera, los trabajadores serán aquellos que crearan y cambiaran de estado los productos que generara la fábrica a través de las etapas. Sin embargo, habrá tres tipos de trabajadores: los azules, encargados de procesar productos azules; los naranjas, encargados de procesar los naranjas; y los rojos, encargados de tomar todos los productos creados y mostrarlos en la parte final.

# 3.5. Asignador de identificador:

Por último, debemos saber que todos los productos deben tener su propio identificador que debe ser diferente al resto de productos, independiente de su color. Por esta razón, debemos tener un actor capaz de asignar un identificador en secuencia para cada producto que se cree en la primera etapa.

# 4.1. Implementación de trabajadores.

Inicialmente, identificamos que se propone que los trabajadores trabajen todos al tiempo (Implementación de *Thread*) y cada uno presente un tipo de espera cuando alguno de los buzones en los que actúa está lleno o vacío, dependiendo de la acción que quiera realizar: los naranjas, realizaran una espera semiactiva usando *Thread.yield();* los azules, optaran por una espera pasiva usando *object.yield()* sobre el buzón que utilizaran; y el rojo, siempre estará pendiente de lo que llega por medio de una espera activa por medio de un bucle *While*. Como podemos evidenciar, todos funcionarán a través de una herencia *Thread* o una implementación *Runnable* y en el programa tendrán los nombres *BlueWorker*, *OrangeWorker* y *RedWorker*, respectivamente.

Previo a detallar los comportamientos, en necesario recordar que el uso de buzones será implementado con *Synchronized,* ya que sabemos que no pueden ser utilizados al tiempo por el tema de sus tamaños limitados.

# 4.1.1. OrangeWorker y BlueWorker.

Antes de cualquier cosa, debemos explicar cómo funciona la construcción de estos trabajadores. En resumen, estos objetos necesitaran un numero de productos a procesar, un buzón donde recoger productos y un buzón donde depositarlos; en caso de ser trabajador de la primera etapa solo necesitara del buzón para depositar.

En este contexto, debemos explicar cómo se tradujo el comportamiento de los trabajadores en *BlueWorkers* y *OrangeWorkers.* Para esta sección, ambos tipos de *Worker* poseen un código que se encarga de revisar en que etapa se encuentran, de tal manera que: si están en la primera etapa, crearan un numero N de objetos y los depositaran en el buzón que tengan como destino; en la segunda etapa, recibirán N productos del buzón de la etapa uno y lo mandaran al siguiente buzón; en la tercera etapa, los trabajadores tomaran N elementos del buzón de la etapa dos y los depositaran en el buzón final (El cual recibe todos los productos de cualquier tipo) para que el trabajador rojo pueda reportarlo como completado. Para completar sus tareas, cuentan con los métodos ***send()*** *y* ***get****()* que se centran en mandar mensajes al buzón o tomar el último elemento depositado en el buzón de manera sincronizada.

# 4.1.2. RedWorker.

Retomando el final del proceso, debemos hablar de la construcción del trabajador final. Este actor se crea utilizando el número de productos que procesan los *OrangeWorkers* y los *BlueWorkers* y el buzón final del cual extraerá los productos para reportarlos. Estructuralmente, se compone de una cola de prioridad que se encargara de revisar los identificadores de cada uno de los productos y los agregara en su posición, de tal manera que se guarde un orden de elementos independiente de su orden de llegada. Por último, este trabajador procesara T\*N productos, donde T es el número de trabajadores existentes y N el número de productos por cada uno. Para completar su tarea, cuenta con un método ***getProduct*()** que se encargara de tomar el último elemento presente en el buzón que se le haya asignado y lo guardara en su cola de prioridad.

# 4.2. Implementación de producto.

Para este punto, decidimos delegar la creación del producto a su respectivo trabajador. Al ser un trabajador, tiene un color asignado y puede pasarlo a la hora de crear un objeto; de esta manera, nos aseguramos de que un *OrangeWorker* cree solo productos naranjas y el *BlueWorker* solo productos azules. Además, cada producto siempre inicia con el estado de creado que será modificado cada vez que es manipulado por un trabajador.

Para lograr su cometido, un producto se implementó con la clase *Product*. De esta manera, el objeto *Product* tiene como métodos: ***updateMessageSent***(), que se encarga de reportar que un producto se fue mandado por un trabajador al siguiente buzón; ***updateMessageReceived***(), que se encargara de reportar que un mensaje fue extraído de un buzón por algún trabajador; y ***transformMessage***(), que se encargara de reportar una transformación a un objeto y el tiempo que demoro en realizarse.

# 4.2.1. GenerateId.

Como mencionamos anteriormente, necesitamos que cada *Product*  tenga un identificador y el único capaz de lograr que no se repitan entre sí y se mantenga una secuencia es el objeto ***GenerateId***. En particular, la creación de los productos será respaldada por esta clase, la cual se encargará asignar un identificador a un producto; específicamente, poseerá un valor entero iniciado en 0 que funcionará como secuencia, de tal manera que cada vez que asigne un identificador la secuencia se aumentará en uno.

Siendo más específicos, esta clase funcionara implementando los siguientes métodos: *createObject*(), que se encargara de crear un objeto del color que le haya mandado por parámetro y un método *getId*() que se encargara de retornar un identificador listo para ser asignado al siguiente objeto.

# 4.3. PRODUCT FACTORY.

Por último, debemos repasar como es la infraestructura de la fábrica y como es su funcionamiento desde su creación. Como bien sabemos, existen tres partes primordiales: la creación, preparación y ejecución. Teniendo las tres etapas anteriores, decidimos crear dos clases ProductFactory y ProductFactoryBuilder, donde el primero se encargará de la preparación y la ejecución, mientras que el segundo se encargará de la creación.

Entrando en detalles, la etapa de creación se encargará de tomar los datos necesarios para crear la fábrica, tales como numero de productos por trabajador, número de trabajadores por etapa y la capacidad máxima de los buzones entre etapas. Por su lado, la etapa de preparación será la parte donde los trabajadores y se les asigna un ron en su respectiva etapa. Por último, la etapa de ejecución se encarga de pedirle a todos los trabajadores que inicien sus tareas.

En términos generales, hay una clase ProductFactoryBuilder que se encargara de tomar los datos necesarios para al ProductFactory.

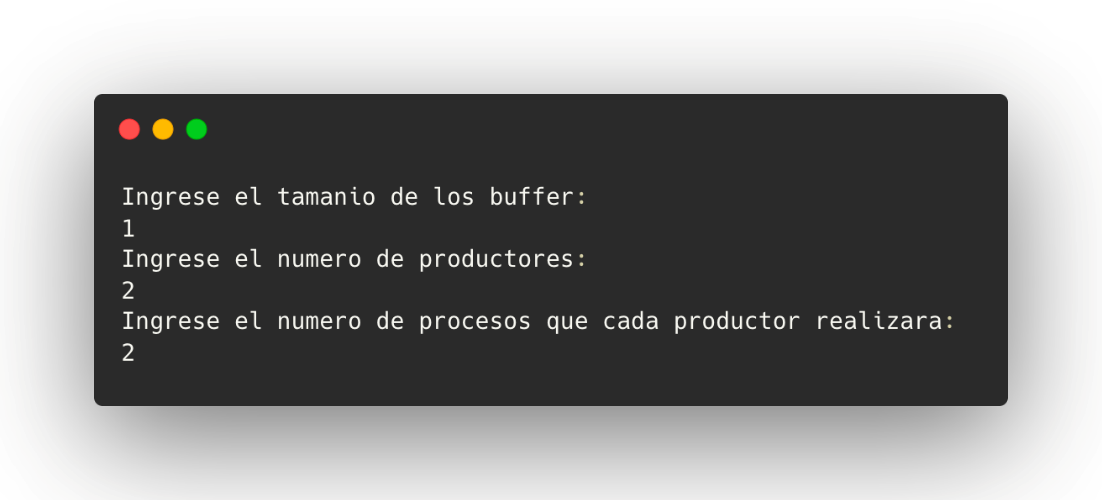
# 4.3.1 Product factory.

En esta parte del proceso, se implementará el proceso de preparación y ejecución. Para la primera fase, existe el método constructor ProductFactory() que se encargara de crear tres buzones, dos finitos y uno infinito, y ejecuta un bucle (Que se repite solo tres veces) que se encargara de crear un numero definido de trabajadores; para esta parte, por cada fase se ejecuta un código diferente para identificar si se debe crear un producto o transportar.

Para la segunda parte, el proceso de ejecución existe un método ***run()*** que se encargara de activar cada uno de los threads (OrangeWorkers, BlueWorkers and RedWorker) utilizando su método ***start()*** con un bucle.

# 5. Validación.

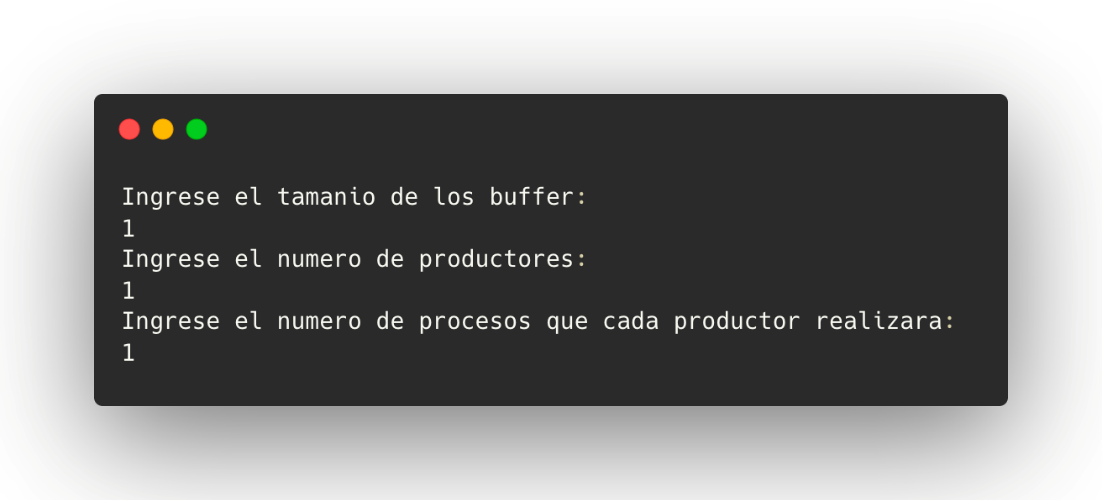
Para demostrar el funcionamiento, el código será sometido a pruebas con diferentes entradas. Inicialmente, utilizaremos una combinación de la siguiente manera. Donde primero se ingresa el tamaño de los buzones, seguido del numeron de trabajadores y por ultimo los productos que cada trabajador procesara.



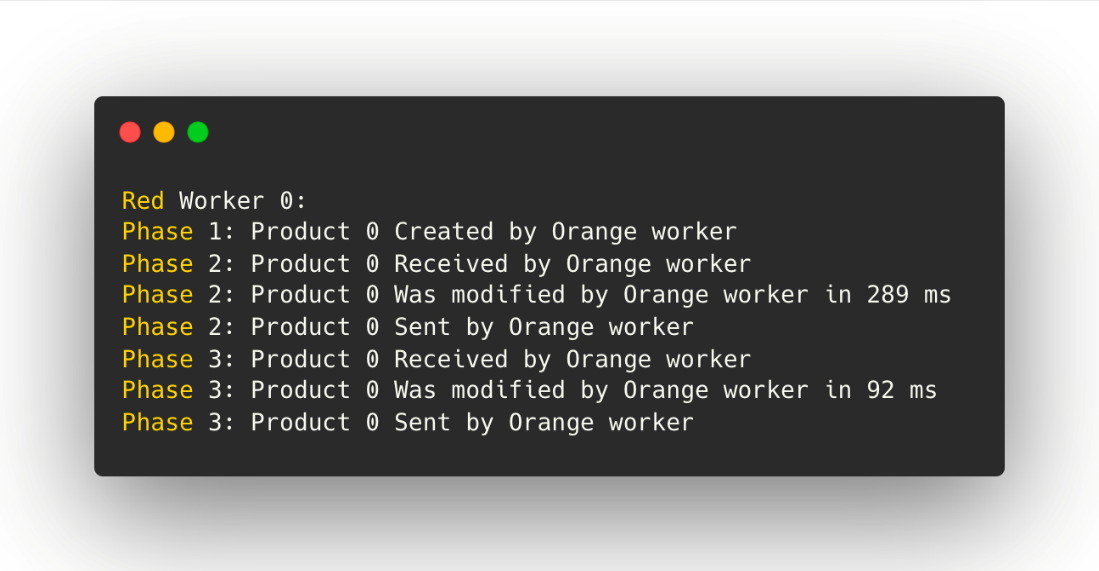
En teoría, se debería mostrar en pantalla un total de cuatro productos, dos azules y dos naranjas (Recordemos que siempre debe haber un único naranja).



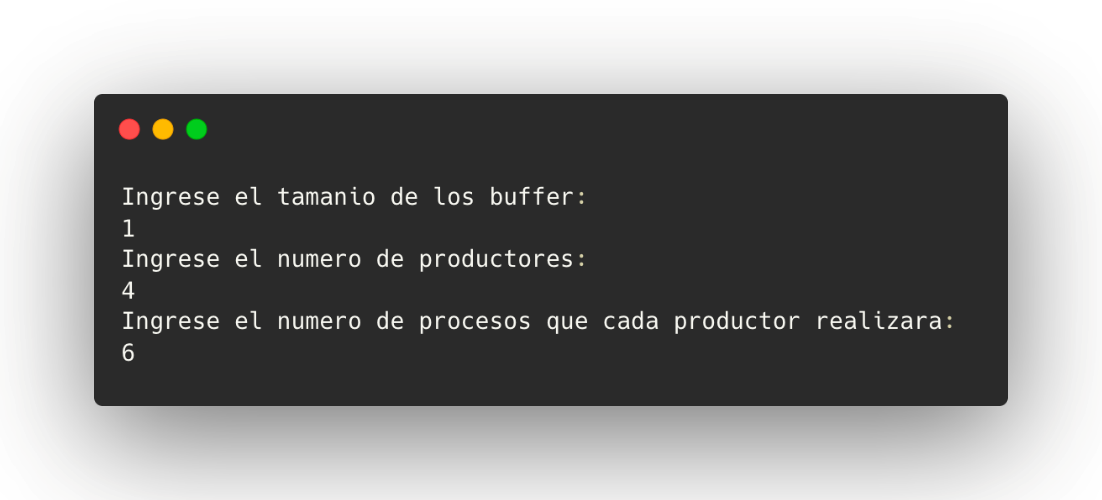
Por otro lado, si tomamos un caso donde solo hay un trabajador, un producto y un máximo de un producto por buffer:



Evidentemente, se espera solo la creación de un producto y su procesamiento completo.



En general, el código funcionara correctamente, pero por temas de espacio en el documento evitaremos mostrar las capturas exactas, sin embargo, podemos afirmar que para un caso (ver siguiente figura) se obtendrá un total de veinticuatro productos, donde solo seis seran naranjas y el resto azules.



# 6. Glosario.

# 6.1 Main.

La clase "Main" es la clase principal de la aplicación en Java. Esta clase se encarga de inicializar el programa y ejecutarlo.

En el método "main", se crea una instancia de la clase "ProductFactoryBuilder" y se almacena en la variable "pfb". A continuación, se llama al método "build" de la variable "pfb" para crear una instancia de la clase "ProductFactory". El objeto resultante se almacena en la variable "pf".

Finalmente, se llama al método "run" del objeto "pf" para ejecutar la lógica principal de la aplicación.

En resumen, esta clase es responsable de iniciar el programa y crear una instancia de la clase "ProductFactory" para ejecutar la lógica de la aplicación.

# 6.2 ProductFactoryBuilder.

La clase "ProductFactoryBuilder" es una clase en Java que se encarga de construir instancias de la clase "ProductFactory". Esta clase lee los parámetros de entrada necesarios para construir una instancia de "ProductFactory" desde la entrada estándar del usuario y luego los utiliza para construir un nuevo objeto "ProductFactory".

El constructor de la clase "ProductFactoryBuilder" no hace nada en particular.

El método "build" es el método principal de la clase. En este método, se crea una instancia de la clase "Scanner" para leer la entrada del usuario. A continuación, se solicita al usuario que ingrese tres parámetros: el tamaño del buffer, el número de productores y el número de procesos que cada productor realizará. Cada uno de estos valores se almacena en una variable correspondiente.

Después de que se han leído los parámetros, se cierra la instancia del objeto "Scanner" y se utiliza el constructor de la clase "ProductFactory" para crear una nueva instancia de esta clase con los valores de los parámetros leídos. Finalmente, se devuelve la instancia creada de "ProductFactory" al llamador del método "build".

En resumen, la clase "ProductFactoryBuilder" es responsable de leer los parámetros de entrada del usuario y construir una nueva instancia de la clase "ProductFactory" utilizando estos valores.

# 6.3 ProductFactory.

La clase ProductFactory es la encargada de crear y controlar el flujo de trabajo de la fábrica de productos. Contiene una lista de hilos (threads) que representan a los trabajadores de la fábrica en cada una de las tres fases de producción, y un método run() que se encarga de iniciar y esperar a que todos los hilos terminen.

El constructor de la clase recibe tres parámetros: bufferSize, stageGroupSize y productCount. Estos parámetros son utilizados para crear los objetos de las clases FiniteMailbox e InfiniteMailbox que representan a los buffers donde se almacenan los productos durante el proceso de producción. Además, se crean los hilos de los trabajadores que estarán involucrados en las tres fases de producción.

En la primera fase de producción, se crea un hilo de la clase OrangeWorker que es responsable de producir los productos y colocarlos en el primer buffer. También se crean varios hilos de la clase BlueWorker que se encargan de tomar los productos del primer buffer y colocarlos de vuelta en el mismo buffer después de haberles aplicado una transformación.

En la segunda fase de producción, se crea otro hilo de OrangeWorker que toma los productos del primer buffer, los procesa y los coloca en el segundo buffer. Se crean nuevamente varios hilos de BlueWorker que toman los productos del primer buffer, los procesan y los colocan en el segundo buffer.

En la tercera fase de producción, se crea otro hilo de OrangeWorker que toma los productos del segundo buffer, los procesa y los coloca en el tercer buffer (un buffer infinito). Nuevamente se crean varios hilos de BlueWorker que toman los productos del segundo buffer, los procesan y los colocan en el tercer buffer.

Por último, se crea un hilo de la clase RedWorker que se encarga de tomar los productos del tercer buffer y realizar la tarea final de la fábrica (en este caso, simplemente imprimir en consola los productos recibidos).

Cada hilo de OrangeWorker y BlueWorker utiliza los objetos FiniteMailbox o InfiniteMailbox para almacenar y procesar los productos. Cada hilo también usa la clase Product para representar los productos que se producen y procesan en la fábrica.

# 6.4 Workers.

# 6.4.1 BlueWorker.

Esta clase BlueWorker representa a un trabajador que recibe y envía productos en una línea de producción. Esta clase se encarga de procesar los productos de tipo "Blue" y transformarlos para enviarlos al siguiente trabajador.

La clase tiene dos constructores: uno para la primera fase de producción donde solo se envían productos, y otro para las siguientes fases de producción donde se reciben productos del trabajador anterior y se envían al siguiente.

La clase extiende la clase Thread, lo que significa que cada instancia de BlueWorker se ejecutará en un hilo separado.

Los métodos más importantes de esta clase son run(), receiveFrom() y sendTo().

El método run() es el método que se ejecuta cuando se inicia el hilo. Este método ejecuta un bucle que se ejecuta contProduct veces (el número de productos que este trabajador debe procesar). Si la fase es 1 (primera fase), el trabajador crea un nuevo producto y lo envía al siguiente trabajador. Si la fase no es 1 (segunda y tercera fase), el trabajador recibe un producto del trabajador anterior, lo transforma y lo envía al siguiente trabajador.

El método receiveFrom() recibe un producto del trabajador anterior y lo devuelve. Si el producto recibido no es de tipo "Blue", el trabajador espera a que llegue un producto de ese tipo antes de continuar.

El método sendTo() envía un producto al trabajador siguiente. Si el buzón de correo del trabajador siguiente está lleno, el trabajador espera a que haya espacio antes de enviar el producto.

En resumen, esta clase representa un trabajador en la línea de producción que procesa productos de tipo "Blue" y los transforma antes de enviarlos al siguiente trabajador.

# 6.4.2 OrangeWorker.

La clase OrangeWorker es una subclase de Thread que representa un hilo de ejecución que se encarga de realizar una tarea específica relacionada con la fabricación de un producto de tipo "Orange". Esta clase tiene varios campos de instancia privados, como mailBox, finalmailbox, contProduct y phase, que se utilizan para almacenar y manipular información específica del hilo.

El constructor de la clase toma dos argumentos: contProduct, que representa la cantidad de productos que se deben fabricar en el hilo, y finalmailbox, que es un buzón de correo compartido donde se envían los productos fabricados por el hilo. El constructor también inicializa el campo phase en 1 para indicar que el hilo está en su primera fase.

La clase OrangeWorker implementa el método run, que se encarga de ejecutar la tarea específica del hilo. El método run utiliza un ciclo for para iterar sobre la cantidad de productos que se deben fabricar. Si el hilo está en su primera fase, el método utiliza el método send para enviar un nuevo producto "Orange" al buzón de correo compartido. Si el hilo está en su segunda fase, el método utiliza el método get para obtener un producto del buzón de correo y luego realiza una serie de operaciones en el producto antes de enviarlo de nuevo al buzón de correo compartido utilizando el método send.

Los métodos send y get son métodos privados de la clase OrangeWorker que se encargan de enviar y recibir productos del buzón de correo, respectivamente. Ambos métodos utilizan la sincronización para asegurarse de que los hilos no intenten acceder al buzón de correo al mismo tiempo.

En resumen, la clase OrangeWorker es una subclase de Thread que se utiliza para fabricar productos de tipo "Orange" y enviarlos a un buzón de correo compartido. La clase utiliza los métodos send y get para enviar y recibir productos del buzón de correo, y el método run para ejecutar la tarea específica del hilo.

# 6.4.3 RedWorker.

La clase RedWorker implementa la interfaz Runnable, lo que indica que sus instancias pueden ser ejecutadas en un hilo aparte. Esta clase tiene la tarea de recibir productos del tipo Product y ordenarlos de manera ascendente en una cola de prioridad. Los productos recibidos contienen un mensaje y un identificador único. Una vez que un producto es recibido, el identificador se utiliza para determinar si se debe imprimir su mensaje o no.

El constructor de RedWorker recibe el número de productos que se esperan recibir y una cola de correo (Mailbox) donde se deben depositar los productos. También recibe el número de trabajadores que están ejecutando esta clase.

El método run() se encarga de recibir los productos de la cola de correo y almacenarlos en la cola de prioridad. Luego, verifica si el primer producto en la cola de prioridad tiene el mismo identificador que el identificador actual. Si es así, se imprime su mensaje y se elimina de la cola de prioridad. El identificador actual se incrementa en uno. Este proceso continúa hasta que se reciben todos los productos esperados y se imprimen todos los mensajes correspondientes.

El método getProduct() se encarga de recibir los productos de la cola de correo. Este método se ejecuta en un bucle infinito hasta que un producto está disponible. Cuando un producto está disponible, se elimina de la cola y se devuelve al llamador. El método notifyAll() se utiliza para notificar a cualquier hilo en espera que hay un producto disponible.

# 6.5 Mailbox.

La clase Mailbox es una interfaz que define los métodos necesarios para implementar una caja de correo genérica. Esta interfaz tiene cuatro métodos:

* isFull(): retorna true si la caja de correo está llena, false en caso contrario.
* isEmpty(): retorna true si la caja de correo está vacía, false en caso contrario.
* send(Product m): envía un objeto del tipo Product a la caja de correo. Retorna true si el envío fue exitoso, false en caso contrario.
* get(): obtiene un objeto del tipo Product de la caja de correo. Si no hay objetos en la caja de correo, se bloquea hasta que llegue un objeto.

Además, la interfaz tiene un quinto método opcional llamado display() que muestra los objetos en la caja de correo en un formato específico.

# 6.5.1 FiniteMailbox.

La clase FiniteMailbox implementa la interfaz Mailbox y representa una caja de correo finita que almacena mensajes de tipo Product. Tiene un buffer interno de mensajes almacenados en un ArrayList, que tiene un tamaño máximo determinado por la variable de instancia 'size'.

Los métodos isFull() e isEmpty() se utilizan para verificar si la caja de correo está llena o vacía, respectivamente.

El método send() agrega un nuevo mensaje a la caja de correo, siempre y cuando el tamaño máximo no se haya alcanzado. Si la caja de correo ya está llena, este método devuelve 'false', de lo contrario, devuelve 'true'.

El método get() se utiliza para recuperar un mensaje aleatorio de la caja de correo. El método elige un índice aleatorio en el ArrayList, recupera el mensaje en ese índice y lo elimina de la lista. Si la caja de correo está vacía, devuelve 'null'.

El método display() simplemente imprime todos los mensajes almacenados en la caja de correo en la consola.

# 6.5.2 InfiniteMailbox.

La clase InfiniteMailbox implementa la interfaz Mailbox<Product> y representa una caja de correos que puede almacenar un número infinito de mensajes. Al igual que la clase FiniteMailbox, esta clase tiene métodos para verificar si la caja de correos está vacía o llena, enviar un mensaje a la caja de correos y recibir el último mensaje de la caja de correos.

El constructor de la clase inicializa la lista de mensajes vacía.

El método isFull() siempre devuelve false porque la caja de correos es infinita y nunca se llena.

El método isEmpty() verifica si la lista de mensajes está vacía y devuelve true si es así.

El método send(Product m) agrega un mensaje a la lista de mensajes y siempre devuelve true porque la caja de correos es infinita y nunca se llena.

El método get() obtiene el último mensaje de la lista de mensajes y lo elimina de la lista. Si la lista está vacía, devuelve null.

El método display() imprime todos los mensajes almacenados en la lista de mensajes.

# 6.6 Product.

La clase "Product" es una clase en Java que representa un producto. Tiene tres atributos: "id", "message" y "color", los cuales representan el identificador del producto, el mensaje del producto y el color del producto, respectivamente.

La clase también implementa la interfaz "Comparable" con el fin de poder comparar objetos de la clase "Product" entre sí.

El constructor de la clase "Product" recibe como parámetros el identificador del producto, el color del producto y una fase. El constructor inicializa los atributos de la clase y también crea un mensaje que indica que el producto ha sido creado por un trabajador de un color específico en una fase específica.

Además, la clase tiene tres métodos: "updateMessageReceived", "updateMessageSent" y "transformMessage", los cuales actualizan el mensaje del producto con información sobre cuándo el producto ha sido recibido, enviado o modificado por un trabajador en una fase específica.

Por último, la clase tiene un método llamado "getType" que devuelve el color del producto.

# 6.7 GenerateId.

La clase GenerateId tiene un método createObject que se encarga de crear objetos de la clase Product. Este método recibe como parámetros el color y la fase del producto que se va a crear. El método genera el id del producto utilizando el método getId, que es un método estático que incrementa un contador interno cada vez que se llama.

El método createObject utiliza el valor de id generado por getId para crear una nueva instancia de Product con el color y la fase dados. La clase GenerateId utiliza un patrón singleton para asegurarse de que solo exista una instancia del contador de id.