Sistemas Operativos. Obligatorio

MIEMBROS

PEDRO SCHEEFFER, SANTIAGO AVELINO, MATIAS DA COSTA

Equipo 8

Planificador de ascensores - Entrega Final

Índice

Primer Avance

Para el primer avance, deben entregar por escrito:

- Una descripción en lenguaje natural del problema desde el punto de vista funcional
- Una identificación preliminar de los recursos y procesos involucrados en la solución
- Una descripción de los distintos criterios de optimización que podrían usarse.
- Un ordenamiento justificado de los criterios y una selección de los primeros que el equipo decida optimizar

Descripción desde el punto de vista funcional

Un programa que atienda llamadas de una cantidad determinada de ascensores y pisos, estos tendrán un par de pulsadores en cada piso con la opción de subir o bajar, además de los pulsadores dentro de la cabina.

Algoritmo de planificación

Para realizar el simulador del manejador de ascensores nosotros decidimos programarlo de una forma que es bastante utilizadas en oficinas como en edificios corrientes. El nombre en general de este algoritmo de funcionamiento del ascensor es “Colectiva ascendente-descendente".

Para poder llevar a cabo la planificación Colectiva ascendente-descendente necesitaremos botoneras colocadas en cada piso. Cada una de estas botoneras contará con dos botones, uno para solicitar la subida, y otro para solicitar la bajada. Dentro del ascensor se maneja la botonera común y corriente con el número de los pisos indicados en ella para que quién suba pueda indicar a que piso tiene intención de ir.

En esta planificación debemos tener en cuenta tres estados en los cuáles se puede encontrar el ascensor, siendo esta información altamente importante para que el planificador pueda decidir cuál es el ascensor más apto para atender el pedido, o como se van a ejecutar las siguientes instrucciones.

Cuando el ascensor se encuentra en subida, el mismo va deteniéndose en todos los pisos marcados desde la cabina interna, y también en los pedidos de piso externos al ascensor que se encuentran marcados como subida, ignorando de esta forma los pedidos de bajada. Al llegar al piso más alto, por encima del último registrado por los pasajeros dentro de la cabina, o por pedidos externos, el ascensor cambia su estado. Dependiendo de si hay pedidos en cola o no, el ascensor cambiará su dirección, o permanecerá quieto durante un período de tiempo. Luego de transcurrido este período de tiempo sin ningún llamado, el planificador le indicará al ascensor que emprenda su trayectoria hacía la planta baja, pasando el estado del ascensor a en bajada.

Podemos ver que, en esta forma de planificación, cuando el ascensor se encuentra en bajada tiene instrucciones similares a cuando está en subida, pero de forma inversa. En bajada, el ascensor va deteniéndose en todos los pisos registrados dentro de la cabina del mismo, y también atiende los pedidos externos desde los pisos, que han pulsado el botón para bajar, ignorando de esta forma aquellos pedidos que sean de subida, hasta que llegue al piso inferior que tenga indicado un pedido de atención. Luego de ese momento el ascensor pasa a estar en un estado de quietud por un período de tiempo, a menos que el planificador considere que tiene

una tarea adecuada para el mismo. Transcurrido cierto tiempo sin llamados de atención, el ascensor se dirige hacía planta baja, cambiando su estado de quieto a en bajada.

Extras de funcionalidad del ascensor paralelas al algoritmo general

El ascensor no podrá exceder la cantidad máxima de 5 ocupantes y un peso máximo de 400 kg.

Recibir llamadas de los pulsadores: El programa deberá estar diseñado para recibir llamadas desde los pulsadores ubicados en los distintos pisos del edificio, y desde los pulsadores ubicados dentro de la cabina. El programa deberá llevar un registro de las llamadas y el estado del ascensor.

Controlar el ascensor: El programa deberá controlar el ascensor de manera que pueda subir o bajar según la llamada que haya recibido y que cumpla con los límites de peso y capacidad de pasajeros. Además, deberá tener en cuenta la dirección de movimiento del ascensor y si hay llamadas pendientes por atender en los pisos que estén en su trayecto.

Gestionar la apertura y cierre de las puertas: El programa deberá gestionar la apertura y cierre automático de las puertas del ascensor una vez que éste se detenga en un piso para permitir el ingreso o salida de los pasajeros.

Monitorear la seguridad del ascensor: El programa deberá tener en cuenta aspectos de seguridad, como el límite de peso y la cantidad máxima de pasajeros permitidos, para evitar cualquier tipo de sobrecarga del ascensor.

Mantener un registro de los viajes realizados: El programa deberá llevar un registro de los viajes realizados por el ascensor, incluyendo la cantidad de pasajeros que lo utilizaron, los pisos en los que se detuvo y la duración del viaje.

Una identificación preliminar de los recursos y procesos involucrados en la solución

Recursos:

- Hardware: se necesitará un equipo de cómputo para desarrollar y ejecutar el programa.
- Software: se necesitará un lenguaje de programación adecuado para desarrollar el programa manejador de ascensores.
- Sensores: se necesitarán sensores de peso y capacidad para monitorear el estado del ascensor.
- Actuadores: se necesitarán actuadores para controlar la apertura y cierre de las puertas del ascensor.

Identificación de procesos que podrían incluirse en nuestra solución:

Recibir llamadas de los pulsadores: El programa debe estar diseñado para recibir llamadas desde los pulsadores ubicados en los distintos pisos del edificio y desde los pulsadores ubicados dentro de la cabina. El programa debe llevar un registro de las llamadas y el estado del ascensor.

Interfaz de usuario: La aplicación también proporciona una interfaz de usuario para que los pasajeros puedan interactuar con el sistema. Esto puede incluir la visualización de información

sobre el estado del ascensor, la selección de pisos deseados y la comunicación con el personal de mantenimiento o emergencia.

Control de llamadas: Este proceso maneja las llamadas realizadas por los usuarios para solicitar un ascensor. La aplicación debe recibir y registrar las solicitudes de cada usuario, incluyendo el piso desde el cual se realiza la llamada y el piso al que se desea ir.

Controlar el ascensor: El programa debe controlar el ascensor de manera que pueda subir o bajar según la llamada que haya recibido y que cumpla con los límites de peso y capacidad de pasajeros. Además, debe tener en cuenta la dirección de movimiento del ascensor y si hay llamadas pendientes por atender en los pisos que estén en su trayecto.

Planificación de rutas: Una vez que se reciben las solicitudes de los usuarios, el sistema debe planificar la ruta óptima para atender todas las solicitudes de manera eficiente. Esto implica determinar el orden de las paradas y minimizar el tiempo de espera de los usuarios.

Gestionar la apertura y cierre de las puertas: El programa debe gestionar la apertura y cierre automático de las puertas del ascensor una vez que éste se detenga en un piso para permitir el ingreso o salida de los pasajeros.

Gestión de prioridades: En ocasiones, podrían surgir situaciones donde se requiera dar prioridad a ciertos usuarios, como personas con discapacidad o emergencias médicas. La aplicación debe tener la capacidad de gestionar estas prioridades y ajustar la planificación de rutas en consecuencia.

Monitorear la seguridad del ascensor: El programa debe tener en cuenta aspectos de seguridad, como el límite de peso y la cantidad máxima de pasajeros permitidos, para evitar cualquier tipo de sobrecarga del ascensor.

Gestión de alarmas y emergencias: La aplicación podría incluir funciones para gestionar alarmas y emergencias. Por ejemplo, puede detectar situaciones anormales, como un ascensor atascado o un mal funcionamiento, y activar alarmas para alertar al personal de mantenimiento o a los pasajeros. También puede proporcionar protocolos de seguridad y procedimientos de evacuación en caso de emergencia.

Monitoreo y mantenimiento: La aplicación también podría incluir funciones de monitoreo y mantenimiento del ascensor. Puede recopilar datos sobre el rendimiento del ascensor, como la frecuencia de uso, tiempos de respuesta, fallas y otros indicadores clave. Estos datos pueden utilizarse para realizar el mantenimiento preventivo y programar reparaciones cuando sea necesario.

Mantener un registro de los viajes realizados: El programa debe llevar un registro de los viajes realizados por el ascensor, incluyendo la cantidad de pasajeros que lo utilizaron, los pisos en los que se detuvo y la duración del viaje

Una descripción de los distintos criterios de optimización que podrían usarse.

Priorizar la planta baja: se podría diseñar el programa de manera que, una vez que el ascensor ha estado inactivo durante un tiempo determinado, éste regrese automáticamente a la planta baja para estar listo para el próximo uso.

Atender a la mayor cantidad de personas: se podría diseñar el programa de manera que el ascensor atienda primero a las solicitudes de los pisos donde hay más personas esperando, para maximizar el número de pasajeros atendidos en cada viaje.

Uso eficiente de recursos: se podría diseñar el programa para que utilice los recursos del sistema de manera eficiente, utilizando algoritmos de programación y estructuras de datos optimizadas para minimizar el consumo de recursos como la memoria y el tiempo de CPU.

Maximizar el uso de la CPU: se podría diseñar el programa de manera que, cuando el ascensor no está en uso, éste realice otras tareas, como monitorear el estado de los sensores y actuadores, actualizar el registro de viajes, o incluso realizar tareas de mantenimiento preventivo.

Priorizar a los clientes VIP: se podría diseñar el programa de manera que, en ciertas situaciones, se les dé prioridad a los clientes VIP, como esperar en un piso específico para recoger a un director que está por llegar, o incluso detenerse en un piso para permitir que un cliente VIP baje antes que otros pasajeros.

Diferentes estrategias dependiendo de la situación: se podría diseñar el programa para que adapte su comportamiento dependiendo de la hora del día, la cantidad de pasajeros que hay en el edificio, o incluso el día de la semana. Por ejemplo, en horas pico, el programa podría priorizar los pisos con mayor cantidad de personas esperando.

Diferentes estrategias para cada ascensor: se podría diseñar el programa para que cada ascensor tenga su propia estrategia de funcionamiento, dependiendo de la ubicación del ascensor en el edificio, la cantidad de pasajeros que atiende y otros factores específicos del ascensor.

Diferentes prioridades según horario: se podría diseñar el programa para que adapte su comportamiento dependiendo del horario, por ejemplo, priorizando la planta baja en horas de la mañana cuando los empleados del edificio están llegando al trabajo, y priorizando los pisos superiores en horas de la tarde cuando los empleados están saliendo.

Minimización del tiempo de espera promedio: La aplicación puede buscar minimizar el tiempo promedio de espera de los usuarios. Esto implica planificar rutas de manera eficiente para atender las solicitudes en el orden óptimo posible, evitando demoras innecesarias.

Equilibrio de carga entre los ascensores: La aplicación puede optimizar la distribución de pasajeros entre los ascensores disponibles para evitar que uno de ellos se sobrecargue excesivamente mientras otros están menos ocupados. Esto ayuda a reducir el tiempo de espera en general y asegura un uso equitativo de los ascensores.

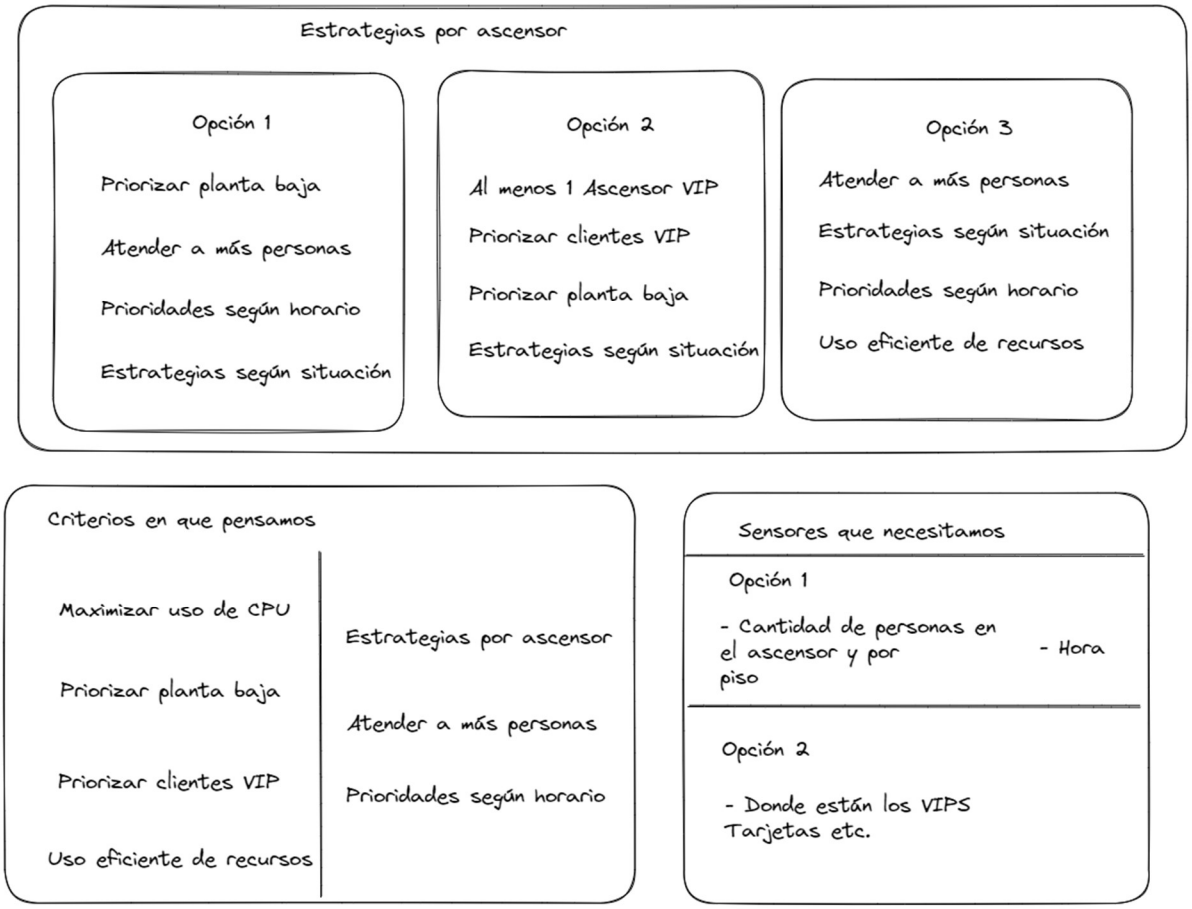
Tiempo de viaje eficiente: Además de minimizar el tiempo de espera, la aplicación puede buscar reducir el tiempo total de viaje de los usuarios. Esto implica planificar rutas que minimicen el número de paradas innecesarias y maximicen la velocidad de desplazamiento entre pisos.

Priorización de llamadas urgentes: La aplicación puede dar prioridad a las llamadas de emergencia o a situaciones urgentes, como llamadas médicas o de seguridad. Esto garantiza una respuesta rápida y eficiente en caso de situaciones críticas.

Un ordenamiento justificado de los criterios y una selección de los primeros que el equipo decida optimizar

Para realizar el punto 4 debemos de tener en cuenta que los criterios justificados son arbitrarios y dependen mucho de la situación específica que solicite el usuario que vaya a implementar el manejador, por lo tanto, esta parte resulta un tanto subjetiva de nuestra parte y va a tener algunas diferencias con respecto a sistemas quizás más convencionales

Decidimos utilizar un criterio de diferentes estrategias para cada ascensor y creamos 3 opciones:



Descripción de las opciones:

Opción 1

El ascensor prioriza la planta baja, atiende a la mayor cantidad de personas posible, ajusta sus prioridades según el horario y utiliza diferentes estrategias según la situación actual del edificio, como la cantidad de personas por piso y la ubicación actual del ascensor.

Opción 2

El principal objetivo de este ascensor es el de priorizar gente con puestos VIP al darles prioridad en cuanto a tiempo de trasladado al utilizar más CPU. Además, cuando la persona llame al ascensor, tendrá más preferencia que otra persona que no sea VIP.

Opción 3

Por otra parte, este ascensor se centrará más en satisfacer las necesidades en general al atender la mayor cantidad de gente posible, sin otorgar prioridad a ciertas personas en específico. También se centrará en el uso eficiente de los recursos, así como dar prioridades a ciertos pisos (los más llenos) según el horario.

Estas tres opciones nos habíamos planteado en la primera entrega con un proceso de trabajo un poco verde, ya que por ejemplo todavía no teníamos en claro siquiera que algoritmo de planificación íbamos a utilizar. Por lo tanto, habíamos ideado tres opciones distintas sobre lo que creíamos que podíamos hacer pero que luego no terminamos utilizando. Hemos decidido dejarlas en el informe del trabajo ya que forman parte del proceso el cuál recorrimos para ir realizando las diferentes partes del trabajo.

Finalmente, y ya teniendo en claro la estructura base del programa del planificador y el algoritmo, pudimos tener mucho más en claro que criterios de optimización vamos a implementar en la práctica de nuestro trabajo.

Criterios de optimización seleccionados para nuestro manejador:

1- Minimización del tiempo de espera promedio: Este criterio se coloca en primer lugar debido a su importancia para los usuarios. Al minimizar el tiempo de espera promedio, se mejora la eficiencia del ascensor y se reduce la frustración de los pasajeros al esperar. Al priorizar este criterio, se busca brindar un servicio rápido y eficiente a todos los usuarios.

2- Equilibrio de carga entre los ascensores: Una vez que se tiene en cuenta la minimización del tiempo de espera promedio, el siguiente criterio es el equilibrio de carga entre los ascensores disponibles. Distribuir de manera equitativa a los pasajeros entre los ascensores ayuda a evitar situaciones en las que un ascensor esté sobrecargado mientras otros están menos ocupados. Esto contribuye a mantener un flujo de tráfico eficiente y reduce los tiempos de espera individuales.

3- Tiempo de viaje eficiente: Después de considerar la equidad en la carga, se busca optimizar el tiempo total de viaje de los pasajeros. Esto implica minimizar el número de paradas innecesarias y maximizar la velocidad de desplazamiento entre pisos. Siempre que no comprometa significativamente los criterios anteriores, este criterio contribuye a una experiencia más fluida y rápida para los usuarios.

4- Priorización de llamadas urgentes (no implementado de momento): A continuación, se da importancia a la priorización de llamadas urgentes, como las llamadas de emergencia o situaciones críticas. Aunque es esencial atender a todos los usuarios de manera justa y eficiente, las situaciones de emergencia requieren una respuesta inmediata y prioridad en el servicio. Este criterio ayuda a garantizar la seguridad y el bienestar de todos los pasajeros.

5- Experiencia del usuario: Aunque es importante, la experiencia del usuario se coloca en último lugar en este ordenamiento. Esto se debe a que los criterios anteriores, como la minimización del tiempo de espera y la eficiencia del viaje, también impactan positivamente en la experiencia del usuario. Sin embargo, una vez que se han abordado los aspectos fundamentales, se busca brindar una interfaz intuitiva, información en tiempo real y sistemas de comunicación para mejorar aún más la satisfacción del usuario.

Este ordenamiento prioriza los criterios clave en el contexto de un ascensor corriente con planificación ascendente-descendente, asegurando un equilibrio entre eficiencia, seguridad y comodidad para los pasajeros.

Sin embargo, de todos modos, dependiendo de las necesidades del entorno o las diferencias de los edificios en los cuales podría encontrarse el ascensor, estos criterios podrían variar de

muchas maneras. Por ejemplo, en un manejador de ascensores para oficina, podríamos tener el piso del jefe, el cuál su llamado podría tener prioridad sobre otros, o podríamos tener un ascensor que este constantemente subiendo y bajando pese a no recibir llamados, como por ejemplo podría ser el mirador de la intendencia (no sabemos realmente si ese es su funcionamiento, pero nos pareció un buen ejemplo de un caso diferente). Al fin y al cabo, todo dependerá de ajustar los criterios a las necesidades y prioridades de los edificios en específico. Nosotros decidimos realizar un caso bastante general ya que creemos que con estos criterios podría ser funcional en prácticamente cualquier edificio salvo que necesite condiciones muy específicas como las anteriormente mencionadas.

Segunda Entrega

Identificación de conceptos y problemas similares de sistemas operativos que puedan aprovechar para el diseño de la solución

Hay varios conceptos y problemas en sistemas operativos que pueden ser útiles para el diseño de soluciones de planificación. Por ejemplo, los algoritmos de planificación de sistemas operativos son utilizados para distribuir recursos entre procesos que los solicitan simultánea y asincrónicamente. Los algoritmos de planificación tienen como objetivo minimizar la inanición de recursos y garantizar la equidad entre los procesos que utilizan los recursos. Algunos algoritmos de planificación comunes en sistemas operativos incluyen First-Come, First-Served (FCFS), Shortest-Job-First, Priority Scheduling, Round-Robin Scheduling y Multilevel Queue Scheduling.

First-Come, First-Served (FCFS): También conocido como First In, First Out (FIFO), es el algoritmo de planificación más simple. FCFS simplemente encola los procesos en el orden en que llegan a la cola de listos.

Shortest-Job-First: Este algoritmo selecciona para ejecución el proceso con el menor tiempo de ejecución.

Priority Scheduling: Este algoritmo asigna prioridades a cada proceso y selecciona para ejecución el proceso con la prioridad más alta.

Round-Robin Scheduling: Este algoritmo asigna una porción de tiempo fija a cada proceso en la cola de listos y se ejecuta en orden circular. Cuando se agota el tiempo asignado a un proceso, pasa al final de la cola y se selecciona el siguiente proceso para su ejecución.

Multilevel Queue Scheduling: Este algoritmo asigna diferentes colas a diferentes tipos de procesos y tiene su propio algoritmo de planificación para cada cola.

Un problema relacionado con nuestra solución que puede ser útil para el diseño de soluciones es el algoritmo del ascensor (también conocido como SCAN), que es un algoritmo de planificación de disco utilizado para determinar el movimiento del brazo y la cabeza del disco al atender solicitudes de lectura y escritura. Este algoritmo se llama así por el comportamiento de un ascensor de edificio, donde el ascensor continúa viajando en su dirección actual (hacia

arriba o hacia abajo) hasta que está vacío, deteniéndose solo para dejar salir a las personas o para recoger a nuevas personas que se dirigen en la misma dirección.

Otro algoritmo existente para el funcionamiento de ascensores se llama planificación “Colectiva descendente”. En este algoritmo, las botoneras colocadas en los pisos intermedios contienen un solo botón para realizar la llamada.

En el modo de subida, el ascensor se detiene en todos los pisos que han sido seleccionados desde dentro de la cabina, pero no responde a ninguna llamada de piso realizada desde fuera, excepto si proviene del piso más alto por encima del último piso registrado por los pasajeros. Una vez que la cabina llega al último piso cuya llamada ha sido registrada, y no se reciben nuevas solicitudes durante un período de tiempo, el ascensor cambia de dirección.

En el modo de bajada, el ascensor se detiene en todos los pisos que han sido seleccionados desde dentro de la cabina, y también responde a las solicitudes de llamada realizadas desde los pisos, siempre y cuando indiquen que desean bajar. Sin embargo, si la cabina está completamente cargada y el ascensor cuenta con un dispositivo de pesaje, no se detendrá en los pisos intermedios. El ascensor continúa bajando hasta llegar al piso más bajo que tenga una solicitud de atención.

Finalmente, luego de tener en cuenta problemas relacionados con nuestra solución, como por ejemplo el SCAN y también teniendo en cuenta otros algoritmos de planificación de ascensores típicos como la planificación “Colectiva descendente”, finalmente nosotros determinamos que nuestro manejador de ascensores funcionará con la planificación del algoritmo “Colectiva ascendente-descendente", la cual se encuentra descrita anteriormente.

Una descripción de al menos tres alternativas para implementar una solución que optimice los criterios seleccionados

Opción 1: Usar los ascensores como recursos, crear el hilo y asignarle un ascensor:

En esta opción, se considera cada ascensor como un recurso que debe ser asignado a un hilo específico. Al iniciar el programa, se crean los hilos correspondientes a los ascensores disponibles. Cuando se recibe una llamada desde un piso, el planificador selecciona un hilo disponible y le asigna el ascensor asociado. El hilo del ascensor es responsable de ejecutar la lógica del ascensor, como decidir si debe recoger o dejar pasajeros, moverse en una dirección específica y responder a las llamadas de los pisos asignados.

Opción 2: Cuando empieza el programa se crean los hilos, estos serán los ascensores:

En esta opción, los hilos se crean al iniciar el programa y cada uno de ellos representa un ascensor. Cada hilo del ascensor tiene su propia lógica para manejar las llamadas y realizar las acciones correspondientes, como recoger y dejar pasajeros, y moverse entre los pisos del edificio. El planificador se encarga de enviar las llamadas a los hilos de los ascensores según la disponibilidad y el estado actual de cada uno.

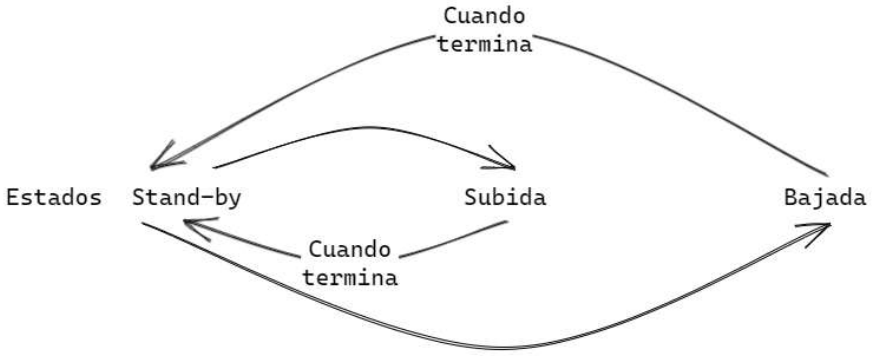
Opción 3: Los hilos representan las personas:

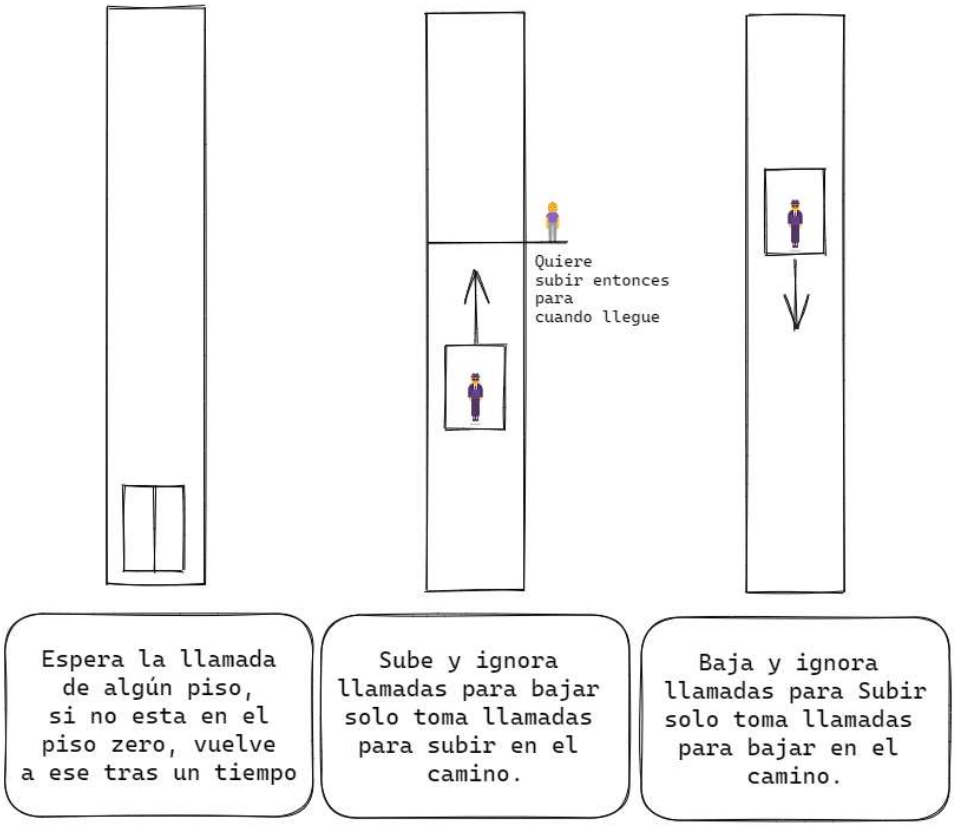
En esta opción, los hilos se utilizan para representar a las personas en el edificio que desean utilizar el ascensor. Cada hilo de persona realiza una llamada al ascensor cuando necesita usarlo. El planificador recibe estas llamadas y selecciona un hilo de ascensor disponible para atender la solicitud de la persona. El hilo del ascensor correspondiente ejecuta su lógica para decidir si debe recoger a la persona en el piso actual o continuar moviéndose hacia otro piso. El objetivo es garantizar un uso eficiente del ascensor y minimizar los tiempos de espera de las personas en los pisos.

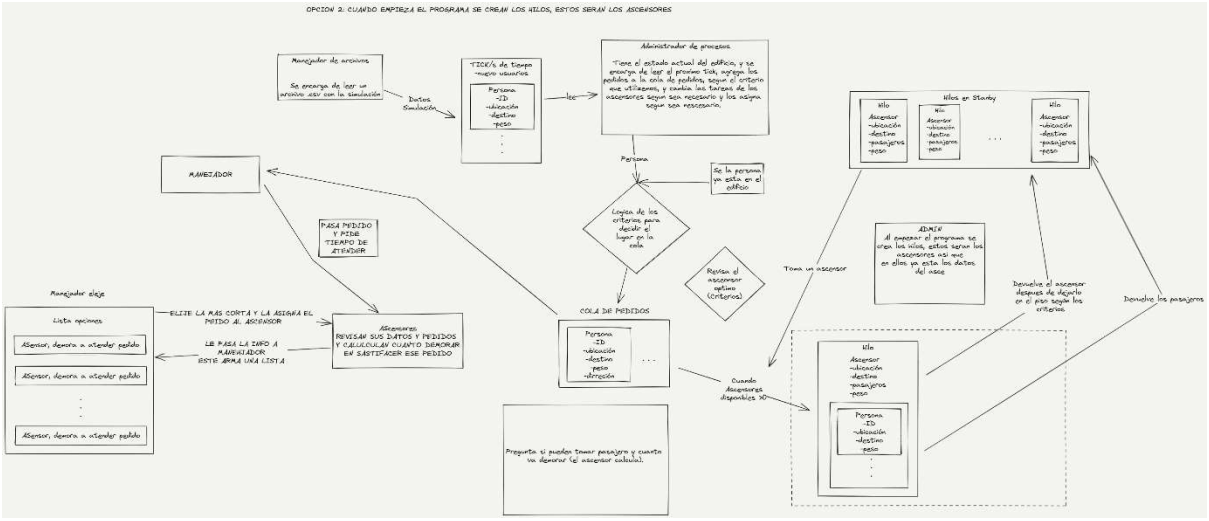
Una selección justificada de la alternativa a implementar

Nosotros decidimos implementar la alternativa en la cual los hilos se crean al iniciar el programa y cada uno de ellos representa un ascensor. Las personas las cuáles puedan tener acceso al ascensor serán instanciadas como un objeto normalmente, el cuál contará con atributos básicos de la persona, como por ejemplo un identificador de la misma, o su peso, el cual es realmente importante en nuestra implementación debido al límite de peso que existe en el ascensor. Decidimos realizar esta implementación primero que nada ya que pensamos que, en el desarrollo del programa, sería mucho más fácil contar con la parte lógica de sincronización de los hilos con semáforos en los ascensores, antes que, en las personas, por ejemplo. También creemos que el crear a las personas como hilos, siendo hilos ya los ascensores sería algo quizás un poco complicado e innecesario al mismo tiempo, la parte de sincronización se puede hacer en los ascensores, y crear un hilo cada vez que una persona quiere ingresar a un ascensor y borrarlo una vez que la persona baja del mismo, nos parece poco lógico al momento de la implementación y deberíamos de tener en cuenta múltiples aspectos, que con nuestra selección se solucionan de manera mucho más sencilla, como por ejemplo la eliminación manual de los hilos. En este programa, los hilos (ascensores) se crean una vez iniciado el programa y no hay necesidad de pensar en eliminar los mismos, pensando por ejemplo en un caso de la vida real, cuando se construye un edificio, una vez creado los ascensores funcionan de manera continua y no es sino cada muy largo período de tiempo que los mismos son sustituidos, por lo tanto, nuestra aplicación coincide con lo que sucede en la realidad, ya que los ascensores se crean al principio, y mientras dure la simulación, los mismos estarán activos y en funcionamiento.

Un bosquejo de la simulación que se efectuará Sistemas Operativos







Tercer Entrega

Para implementar el manejador de ascensores decidimos utilizar Java

La decisión del lenguaje de programación a utilizar fue en gran parte ya que en clase hicimos ejercicios de práctica con Java y nos acostumbramos a trabajar con hilos con este mismo.

También, luego de haber manejado diferentes opciones como C# o Python, llegamos a la conclusión de que la implementación de hilos, semáforos y todo lo relacionado a estos, tiene un manejo bastante más intuitivo en Java que en los lenguajes mencionados anteriormente. Esto se debe a las librerías que ya vienen incluidas en Java, como por ejemplo la interfaz Runnable o la clase Thread.

La clase Thread y la interfaz Runnable son dos elementos clave en Java para trabajar con hilos y proporcionan facilidades para la creación y ejecución de hilos en un programa.

Clase Thread: La clase Thread es una clase predefinida en Java que proporciona métodos y funcionalidades para trabajar con hilos. Se puede utilizar de dos maneras:

a) Extendiendo la clase Thread: Se puede crear una nueva clase que herede de Thread y luego sobrescribir el método run(). Dentro de run(), se definen las acciones que se buscan que realice el hilo. Luego, se puede crear una instancia de la clase y llamar al método start() para iniciar la ejecución del hilo.
b) Creando una instancia de Thread y pasando una instancia de Runnable: Se puede crear una instancia de la clase Thread, pasando un objeto que implemente la interfaz Runnable como argumento al constructor de Thread. La interfaz Runnable define un único método llamado run(), que se debe implementar con las acciones que se buscan que realice el hilo. Luego, se debe llamar al método start() en la instancia de Thread para comenzar la ejecución del hilo.

Interfaz Runnable: La interfaz Runnable es una interfaz funcional en Java que proporciona un solo método llamado run(). Esta interfaz es implementada por cualquier clase que desee definir acciones a ejecutar en un hilo. Al implementar Runnable, se debe proporcionar una implementación para el método run(), que contiene el código que se busca que el hilo ejecute.

Al utilizar la interfaz Runnable, se puede separar la lógica de ejecución del hilo de la lógica de la clase principal. Esto facilita la reutilización del código y mejora la modularidad del programa.

En nuestro programa, la clase ascensor implementa la interfaz Runnable, y de esta forma, desde el ascensor podemos acceder a toda la lógica del funcionamiento y manejo de hilos. Los ascensores son quienes se sincronizan entre sí y hay parte de esta lógica también en el planificador.

Explicación del programa del ascensor clase por clase

Clase Persona

lib > Persona.java > Persona

```java
public class Persona {
    int tick; // tick
    int id; // id
    int ubicacion; // ubication
    int destino; // destination
    int peso; // weight
    Ascensor ascensor; // lift

    public Persona(int tick, int id, int ubicacion, int destino, int peso){
        this.tick = tick;
        this.id = id;
        this.ubicacion = ubicacion;
        this.destino = destino;
        this.peso = peso;
        this.ascensor = null;
    }
    Codeium: Refactor | Explain | Generate Javadoc
    public String toString(){
        String result = Integer.toString(tick) +
                        Integer.toString(id) +
                        Integer.toString(ubicacion) +
                        Integer.toString(destino) +
                        Integer.toString(peso);
        return result;
    }
}
```

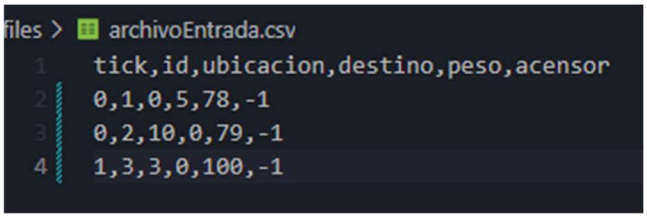
En nuestra implementación del manejador de ascensores nosotros decidimos que las personas quienes van a utilizar los ascensores sean un objeto bastante simple, sin ningún tipo de uso de hilos en su implementación.

La clase persona no cuenta en si misma con ningún método en nuestro diseño del código, la misma solo tiene atributos y su constructor.

- El atributo tick indica en qué momento de la ejecución la persona está realizando la llamada al ascensor. En la ejecución del programa cada tick representa una unidad de tiempo, el ascensor demora exactamente un tick en cambiar de piso. Lo utilizamos como una forma de representar el tiempo que pasa durante la ejecución de forma sencilla.
- El id de la persona es simplemente una forma de identificar a la persona que va a querer ingresar al ascensor, distinguiendo así los diferentes objetos persona entre sí. Es una forma sencilla para clarificar que es lo que está pasando en la ejecución y quien exactamente está llamando al ascensor y hacía donde quiere ir.
- El atributo ubicación indica desde donde está realizando el llamado la persona.
- El atributo destino indica hacía que piso quiere ir la persona.
- El atributo peso indica el peso de la persona correspondiente, si el peso de la persona es superior a la capacidad máxima del ascensor, este mismo no atiende a la llamada de esta persona, y tendrá que esperar por otro ascensor o a que la capacidad del anterior pueda soportar su peso finalmente.

- También la persona tiene un atributo de tipo Ascensor en su código, esto es simplemente para que el objeto persona pueda entender la existencia de un ascensor en el programa y que sea más fácil su interacción. El mismo se inicializa como null cuando se crea al objeto persona, y esto se debe a que cuando la persona se crea en el programa no tiene ningún ascensor asociado, sino que esto sucederá más adelante en partes del código que manejen la lógica del programa.

Archivo de entrada CSV



El programa recibe un archivo de entrada csv para poder procesar gran cantidad de llamadas y probar como se desempeñaría el programa con esta densidad de pedidos. Podríamos cargar un archivo con el siguiente formato con miles de personas y de esta forma sería mucho más fácil ver como se desempeña el manejador.

Los valores se corresponden con los parámetros de la clase persona.

Por ejemplo, supongamos el caso en el que los datos del archivo toman los siguientes valores:

0,1,0,5,78,-1

El primer 0 indica que la persona en el tick 0 de la simulación está realizando un pedido.

El primer uno representa la ID de la persona, el cual solo se utiliza para la identificación de la misma persona.

El segundo 0 indica el piso del edificio desde el cuál la persona está realizando el pedido, en este caso realiza el pedido desde el piso 0.

El primer 5 indica el piso del edificio al cuál la persona quiere ir, en este caso la persona quiere ir hasta el piso 5.

La persona en este caso quiere ir desde el piso 0 hasta el piso 5.

El 78 indica el peso de la persona en kilogramos, el cuál es bastante importante para el desarrollo del programa porque como ya mencionamos anteriormente, si el peso de la persona, sumado al de quienes ya se encuentra en el ascensor supera la capacidad del ascensor, este ignora la llamada del mismo, y se busca asignar otro ascensor o en su defecto que otro ascensor pase por él.

Supongamos que el ascensor está vacío, en ese caso la persona ingresaría al ascensor y comenzaría a dirigirse hacia su destino. Por otro lado, supongamos que en el ascensor ya hay 6 personas, y que la suma total del peso de las personas que se encuentran dentro sea de 350 kg. La capacidad máxima del ascensor puede variar según lo definamos en el código, pero supongamos también que en este caso la capacidad máxima es de 400 kg. Cuando el ascensor pase por donde se encuentra está persona va a realizar el cálculo de peso persona + sumatoria

del peso total dentro del ascensor que en este caso daría un total de 425kg, valor el cual es mayor a 400kg y por lo tanto el ascensor haría caso omiso hasta que tenga una capacidad adecuada para poder recibir a la persona.

También creamos un script en Python csvCreator.py, que está en la carpeta scripts, en este modificando las variables podemos crear varios casos de pruebas distintos, modificando todos los valores descriptos anteriormente, excluyendo ascensor.

Clase Planificador

```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

public class Planificador {
    String pathDefault = "files/archivoEntrada.csv";
    int tick = 0;
    int ticksTotales = 2000;
    int cantidadAscensores = 1;
    List<Persona> todasLasPersonas = new ArrayList<>();
    List<Persona> esperandoAscensor = new ArrayList<>();
    static Planificador _instancPlanificador; // queremos solo un planificado

    ReentrantLock lockLevantarPasajero = new ReentrantLock(); // Locks para Threads
    // Semaforo ascensores
    Semaphore semaforoTick = new Semaphore(permits:0);

    public Planificador() {
        // Se crea y le el archivo csv
        _instancPlanificador = this;
        FileManager fm = new FileManager();
        todasLasPersonas = fm.csvToPerson(pathDefault, skipFirstLine:true); // TODO mover a una variable para no recalcular
    }

    Codeium: Refactor | Explain | Generate Javadoc
    public static Planificador GetPlanificador() {
        if (Planificador._instancPlanificador == null) {
            return Planificador._instancPlanificador = new Planificador();
        } else {
            return Planificador._instancPlanificador;
        }
    }

}
```

La clase planificador es aquella la cuál procesa la gran mayor parte de la información para el funcionamiento del programa.

Podemos observar que la misma es la encargada de leer y procesar los datos del archivo CSV y guardarlos en las correspondientes listas de personas, como pueden ser “todasLasPersonas” o “esperandoAscensor”. En el constructor se cargan las personas a la lista de todasLasPersonas”.

El planificador también tiene un contador de tick personal que va simulando cada paso de la ejecución del programa. Cada tick representa un segundo y la cantidad de ticks totales pensados para nuestro programa es de una cantidad total de 2000 ticks.

El planificador también tiene un atributo entero que le sirve al planificador para conocer la cantidad de ascensores que se encuentran disponibles para la ejecución del programa.

A la lista de personas “esperandoAscensor” se las agrega a la misma en el momento en el tick de tiempo que las mismas realizan un pedido, y son removidas de esta una vez que llegan a su destino.

Como aclaración, la clase Planificador esta creada como un Singleton. Esto se debe a que no nos interesa ni queremos que se genere más de una instancia de la clase Planificador.

```java
    public void Simular() {

        // Empezamos los ASCENSORES
        for (int i = 0; i < cantidadAscensores; i++) {
            Thread ascensorThread = new Thread(new Ascensor(i));
            ascensorThread.start();
        }
        // TODO imprimir informacion de las personsa, de cada ascensor

        // Empieza Simulacion
        for (tick = 0; tick < ticksTotales; tick++) {
            try {
                System.out.println("------- Tick: " + tick + " -------");
                esperandoAscensor.addAll(procesarPersonas(todasLasPersonas, tick));
                // Log de consola
                System.out.println(x:"Entraron en el edificio ");
                for (Persona persona : esperandoAscensor) {
                    System.out.println("Persona " + persona.id + " realiza pedido desde " +
                    persona.ubicacion + " hasta " + persona.destino
                    );
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                semaforoTick.release(cantidadAscensores); // Release permits for all Ascensor
                threads
                try {
                    Thread.sleep(millis:200);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

En Simular() se encarga de manejar los recursos, este empieza los threads y cada a tick carga nuevos pasajeros a la lista esperandoAscensor, luego le da el turno a los ascensores para hacer su trabajo, para esto le suma al semáforoTick la cantidad de ascensores que tenemos, luego espera que los ascensores terminen su trabajo y sumen a semaforoAscensores la cantidad e ascensores que hay, para finalmente repetirse el ciclo.

```java

    Codeium: Refactor | Explain | Generate Javadoc
    public List<Persona> procesarPersonas(List<Persona> todasLasPersonas, int tick) {
        List<Persona> entranAhora = new ArrayList<Persona>();
        for (Persona persona : todasLasPersonas) {
            if (persona.tick == tick) {
                entranAhora.add(persona);
                if (todasLasPersonas.size() == 0) {
                    break;
                }
            } else if (persona.tick > tick) {
                break;
            }
        }
        return entranAhora;
    }
}

```

Lee la lista de toda las personas, toma las que entran en el tick actual y luego las devuelve en otra lista.

Ascensor.java

La clase Ascensor se encarga de simular un ascensor y gestionar los hilos del programa. Contiene información y funcionalidades relacionadas con el ascensor.

```
lib > Ascensor.java > Ascensor > _cantidadPersonasMaximas
1   import java.util.ArrayList;
2   import java.util.Iterator;
3   import java.util.List;
4
5   class Ascensor implements Runnable {
6       // Valores Ascensor
7       int _id;
8       EstadoAscensor estado = EstadoAscensor.DETENIDO; // DETENIDO, SUBIENDO, BAJANDO, ENTRANDO
9       public List<Persona> pasajeros = new ArrayList<Persona>();
10      private int _pesoActualkg = 0;
11      private int _pesoMaximokg = 400; // kg
12      private int _cantidadPersonasMaximas = 5;
13      private int _ubicacion = 0;
14      private int _destino;
15
16      Ascensor(int Id) {
17          _id = Id;
18      }
```

En la siguiente captura de código, se muestra la definición de las variables y la función constructora. El constructor asigna un ID al ascensor para facilitar su identificación en los registros. Además, se agrega un estado al ascensor para controlar su comportamiento: detenido, bajando o subiendo.

```
@Override
public void run() {
    while (!Thread.currentThread().isInterrupted()) {
        try {
            Planificador.GetPlanificador().semaforoTick.acquire();
            bajarPasajeros();
            // semaforo que controla el uso de los tiks
            System.out.println("Ascensor: " + _id +
                    " Piso: " + _ubicacion + " Destino: " + _destino +
                    " Pasajeros: " + pasajeros.size() + " Peso: " + _pesoActualkg + " kg");
            Planificador.GetPlanificador().lockLevantarPasajero.lock();
            switch (estado) {
                case DETENIDO:
                    Detendio();
                    break;
                case SUBIENDO:
                    Subiendo();
                    break;
                case BAJANDO:
                    Bajando();
                    break;
                default:
                    break;
            }
            Planificador.GetPlanificador().lockLevantarPasajero.unlock();
            // TODO agregar if donde reviso si baja gente o sube, si lo hace el ascensor no
            // se mueve
            irDestino();
```

La función run() es el núcleo de la lógica del ascensor. Al inicio, adquiere un semáforo (semaforoTick) para comenzar a realizar sus tareas. Luego, baja a los pasajeros, imprime información y levanta pasajeros según el estado del ascensor. Se utiliza un monitor (lockLevantarPasajero) para asegurar que solo un ascensor a la vez pueda acceder a la lista de pasajeros en el planificador. Una vez que los pasajeros son levantados, se libera el monitor. Finalmente, se llama a la función irDestino() para moverse a un nuevo piso.

```
// DETENIDO, revisa si hay pedidos si los hay y los puede tomar y cambia su
// estado para su proximo tick
Codeium: Refactor | Explain
public void Detendio() {
    // si tiene pasajeros cambia el destino para el del pasajero
    if (!pasajeros.isEmpty()) {
        actualizarDestino(pasajeros.get(index:0), estado);
        if (pasajeros.get(index:0).destino > this._ubicacion) { // cambio el estado del ascensor
            this.estado = EstadoAscensor.SUBIENDO;
        } else {
            this.estado = EstadoAscensor.BAJANDO;
        }
    } else {
        if (Planificador.GetPlanificador().esperandoAscensor.size() == 0) {
            if (_destino != 0 || _ubicacion != 0) {
                _destino = 0;
            }
            estado = EstadoAscensor.DETENIDO;
        }
    }
    // revisa los pedidos actuales y lev anta los pasajeros
    levantarPasajeros(this.estado);
    // termina su turno
}
```

Detendio() es de los 3 métodos según estado el más complejo de los 3, este realiza las La función Detenido() es la más compleja de las tres funciones según el estado del ascensor. Realiza las siguientes tareas:

1. Verifica si el ascensor aún tiene pasajeros cuando se detuvo. En caso afirmativo, actualiza su destino y cambia de dirección.
2. Si no tiene pasajeros y no hay pasajeros esperando, se mueve al primer piso y cambia su estado.
3. Si no se cumplen ninguna de las condiciones anteriores, levanta pasajeros.

```java
private void levantarPasajeros(EstadoAscensor estado) {
    Planificador planificador = Planificador.GetPlanificador();
    switch (estado) {
        case DETENIDO:
            procesarPasajeros(planificador.esperandoAscensor, estado);
            if (this.pasajeros.isEmpty()) {
                buscarPasajerosOtrosPisos();
            }
            break;
        case SUBIENDO:
        case BAJANDO:
            procesarPasajeros(planificador.esperandoAscensor, estado);
            break;
        default:
            break;
    }
}
```

La función levantarPasajeros() se encarga de levantar pasajeros según el estado del ascensor. En los estados "subiendo" y "bajando", simplemente se suben los pasajeros en el piso actual. En el caso de "detenido", primero busca pasajeros en el piso actual y, si no hay ninguno, busca en los demás pisos para atender el pedido más antiguo.

```java
private void procesarPasajeros(List<Persona> personas, EstadoAscensor estado) {
    Planificador planificador = Planificador.GetPlanificador();
    Iterator<Persona> iterator = planificador.esperandoAscensor.iterator();

    while (iterator.hasNext()) {
        Persona persona = iterator.next();
        if (puedeSubir(persona, estado)) {
            pasajeros.add(persona);
            _pesoActualkg = _pesoActualkg + persona.peso;
            actualizarPasajeroSubida(persona);
            actualizarDestino(persona, estado);
            iterator.remove();
        }
    }
}
```

La función procesarPasajeros() revisa si hay pasajeros en el piso donde se encuentra el ascensor y se asegura de que puedan subir. Si pueden, se los sube y se actualiza la información del ascensor y del pasajero.

La función puedeSubir() verifica la capacidad del ascensor y si puede levantar personas en ese momento. Luego, revisa si la persona se encuentra en el mismo piso que el ascensor.

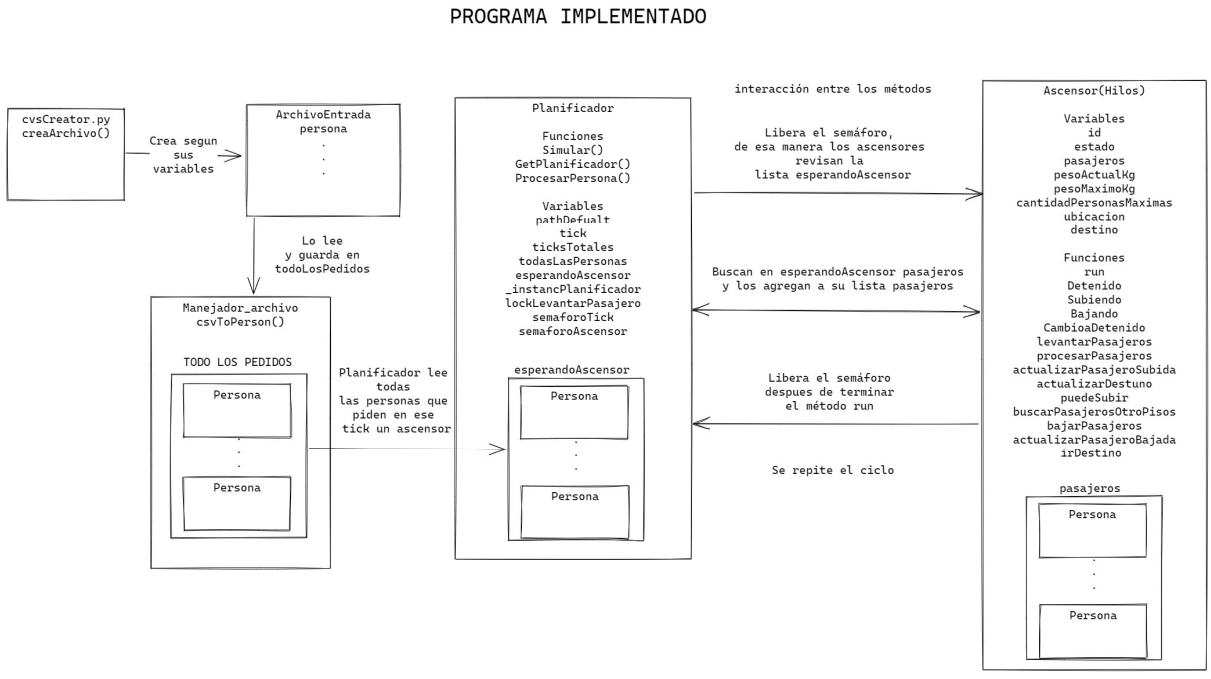
La función actualizarPasajeroSubida() actualiza la variable "ascensor" de la persona. Esto permite que, en la función buscarPasajerosOtrosPisos(), cuando el ascensor busca a una persona en otro piso y verifica esa persona, pueda saber si ya hay un ascensor asignado a ella y así ignorarla y seguir buscando su próximo pasajero.

La función actualizarDestino() se utiliza cuando un nuevo pasajero tiene un piso de destino mayor al destino actual del ascensor.

```java
    private void irDestino() {
        String text = ("Movimiento Ascensor: " + _id + " Piso: " + this.
        _ubicacion + " Destino: " + _destino);
        if (_destino != _ubicacion) {
            if (_destino > _ubicacion) {
                _ubicacion++;
            } else {
                _ubicacion--;
            }
        }else if (!pasajeros.isEmpty()) {
            actualizarDestino(pasajeros.get(index:0), this.estado);
        }
        System.out.println(text + " --> Nuevo Piso: " + this._ubicacion);
    }
}
```

Finalmente, la función irDestino() se encarga de mover el ascensor desde su piso actual hasta su destino. Si ya se encuentra en el destino, verifica si todavía hay pasajeros.

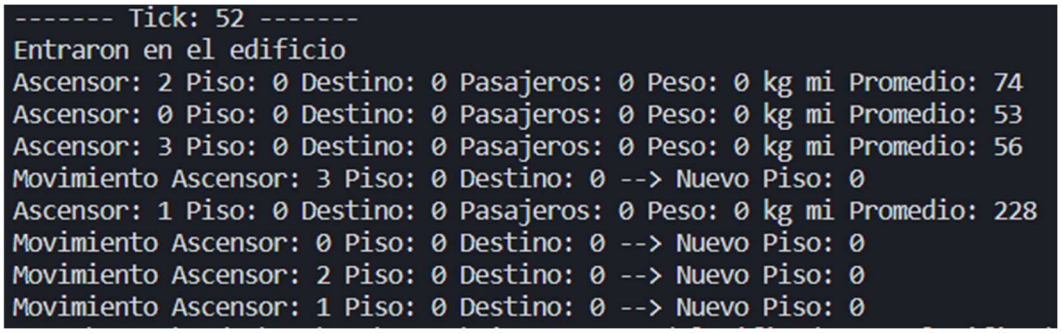
Diagramas que muestren los objetos involucrados en la solución, que variables y operaciones definen cada uno y que interacción se da entre los mismos.



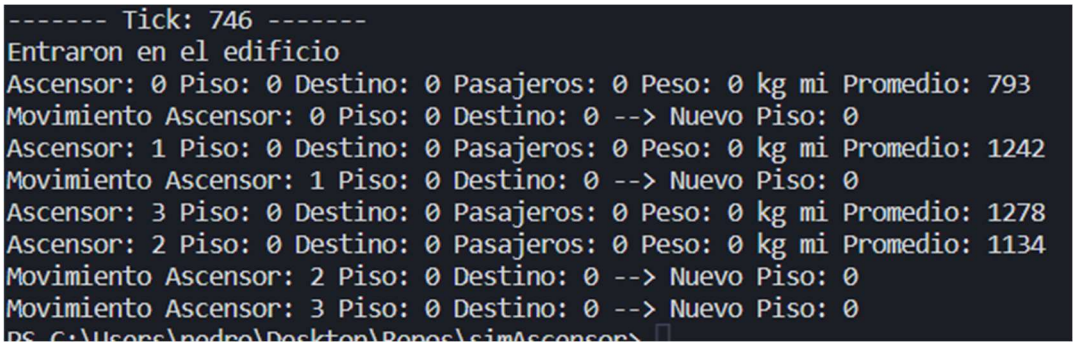
Los resultados de la simulación realizada con su correspondiente análisis y conclusión, ¿se alcanzó lo esperado? ¿por qué no? justificar.

Tras hacer algunas simulaciones y revisando el comportamiento de los ascensores podemos confirmar que estos funcionan como lo esperado, en general los 4 ascensores terminan teniendo promedios de atendimiento bastante similares, con algunos casi siempre 1 de los 4 teniendo un promedio un poco menor, la mitad o menos de los demás, no pudimos hacer un análisis profundo del porqué.

Ahora hablando de la simulación esperada en el planteamiento original, no pudimos implementar los criterios más sofisticados para el atendimiento, principalmente por falta de tiempo para hacerlo.

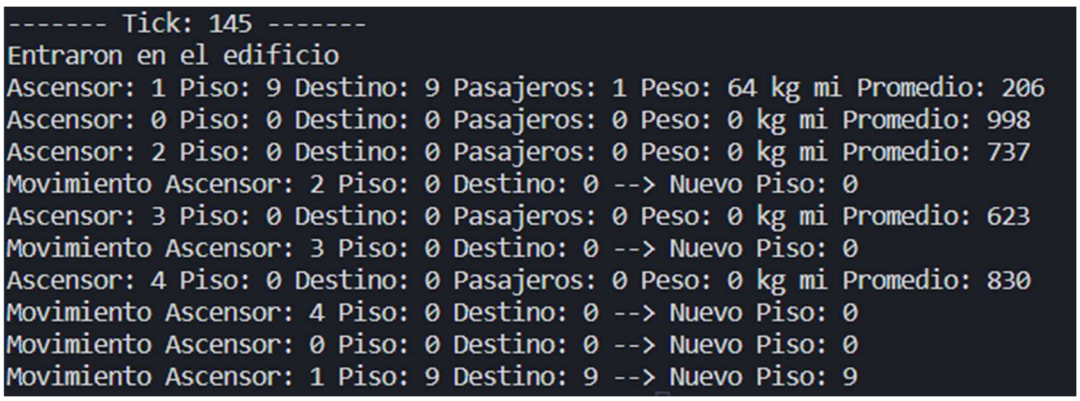


Resultado de una simulación. Los ascensores ya atendieron a todos los pasajeros.

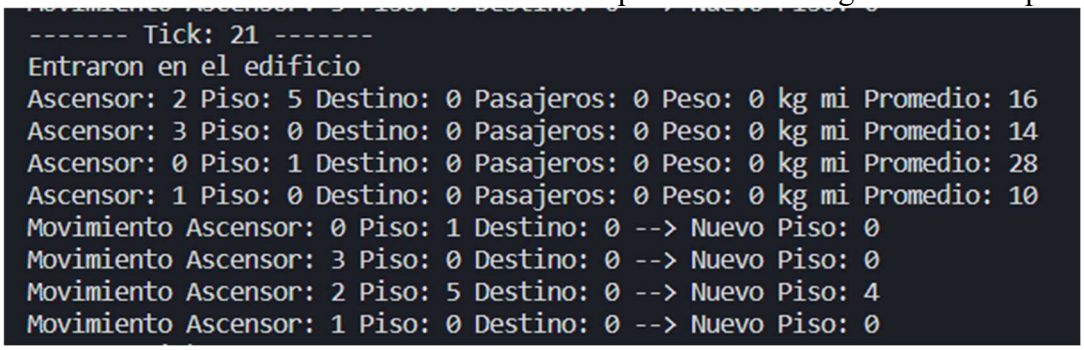


Otro ejemplo con un total de 292 pasajeros que entraron en 100 ticks con una cantidad máxima de 5 personas por tick. Podemos ver que el promedio de tick en que la persona entro al edificio y tick donde llego a su piso aumenta considerablemente.

Con la misma entrada pero con 5 ascensores podemos ver una considerable reducción en tiempo de espera.



Con un caso más razonable de 7 personas conseguimos tiempos menores



Conclusiones del trabajo

Para comenzar a realizar el trabajo, decidimos desde un principio, investigar como funcionaban la mayor parte de edificios comunes y corrientes. Tambien, en el proceso de investigación, averiguamos acerca de soluciones y problemas similares al manejador de ascensores, como por ejemplo lo es el SCAN o el también denominado algoritmo del ascensor. De esta forma, encontramos una gran cantidad de algoritmos de planificación de entre los que todos nos quedamos con el ascendente-descendente, debido a que nos pareció interesante el pensar como sería llevar este algoritmo tan usado a nuestro código.

Una vez definido el algoritmo de planificación que iba a utilizar nuestro sistema tuvimos que establecer un ordenamiento de los criterios que el planificador de ascensores debería de seguir. Para esto en un principio manejamos y analizamos la opción de un montón de criterios de planificación, los cuáles se pueden ver en la sección de “Una descripción de los distintos criterios de optimización que podrían usarse”. En un principio no teníamos muy claro cuáles serían los criterios que mejor se adaptarían a nuestro algoritmo de planificación, pero luego de investigar un poco más de que se trataban los mismos, seleccionamos los siguientes criterios para nuestro manejador de ascensores:

1- Minimización del tiempo de espera promedio

2- Equilibrio de carga entre los ascensores

3- Tiempo de viaje eficiente

4- Priorización de llamadas urgentes (no implementado de momento)

5- Experiencia del usuario

En un caso ideal de un ascensor con el algoritmo de planificación ascendente-descendente estos hubieran sido unos excelentes criterios a seguir, ya que satisface con casi todos los puntos para que la experiencia del usuario sea buena y versátil, ya que equilibrando la carga entre los ascensores y buscando repartir el equilibrio de carga de los mismos, son criterios que siempre favorecen al usuario. Finalmente, en nuestra implementación del programa no nos logramos ceñir a los criterios de planificación establecidos previamente, por cuestiones de tiempo. Nuestro manejador de ascensores sigue el algoritmo ascendente-descendente, pero su

implementación de que ascensor atiende a que persona y a que persona se atiende primero utiliza un algoritmo de planificación bastante simple, el cuál es el First Come First Served, o también conocido como FIFO. En nuestro programa el orden de como los ascensores atienden los llamados de las personas se basa en quien llamo primero al ascensor, aunque por ejemplo si hay un pedido del piso 8 en primera instancia, y el ascensor está subiendo y va por el piso 3 y recibe un pedido desde el piso 6, si atenderá a la persona que se encuentra en el sexto piso para luego atender a quien está en el octavo. Los recursos en nuestro caso son las personas esperando y la competencia entre hilos se da entre los ascensores, el que consigue acceder a los recursos, es decir a las personas, es el ascensor que primero llega a donde está se encuentra.

En conclusión, de esta parte en concreto, no logramos cumplir con los criterios de optimización esperados, pero si logramos tener un programa que maneja los ascensores mediante la sincronización de hilos de una forma coherente, y por el camino en el que íbamos de haber contado con más tiempo podríamos haber logrado llegar a cumplir esos criterios.

Por lo tanto, podemos decir que los aspectos a mejorar sería el ceñirse mejor a los criterios de optimización, controlando el balance de carga entre los ascensores y de esta forma buscar el minimizar el tiempo de espera promedio para las personas.

También nos faltó implementar una forma de priorizar llamadas urgentes, la cual podríamos haber implementado creando quizás una persona que tuviera prioridad en los llamados, o generando un método especial en el ascensor que al ser llamado de esa manera el ascensor ignore otros llamados y le dé prioridad total a ese pedido.

Esos serían los siguientes pasos a realizar y cambios en la implementación a futuro.

Bibliografía

- Codeium herramienta para desarrolladores, ayuda con “boilerplate code” https://codeium.com

-colaboradores de Wikipedia. (2023). Ascensor. *Wikipedia, la enciclopedia libre*. https://es.wikipedia.org/wiki/Ascensor

- K, H. (2018, 26 octubre). *La ciencia oculta de los ascensores*. Revista del Ascensor. https://revdelascensor.com/2018/10/26/la-ciencia-oculta-de-los-ascensores/

- Torres, A., & Amaya, J. (2021). *Diseño de un sistema de control para ascensores con análisis de sus variables energéticas en la nube utilizando Machine Learning.* (Estudiantes). UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS FACULTAD TECNOLÓGICA.

- EHU. (s. f.). *Algoritmos de planificación*. Recuperado 18 de junio de 2023, de https://lsi.vc.ehu.eus/pablogn/docencia/manuales/SO/TemasSOuJaen/PLANIFICACIONDEPROCESOS/6AlgoritmosdePlanificacionI.htm

- Williams, L. (2023). Priority Scheduling Algorithm: Preemptive, Non-Preemptive EXAMPLE. *Guru99*. https://www.guru99.com/priority-scheduling-program.html

- GeeksforGeeks. (2023). Multilevel Queue MLQ CPU Scheduling. *GeeksforGeeks*. https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/

- *Thread (Java Platform SE 8)*. (2023, 5 abril). https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

- *Runnable (Java Platform SE 8)*. (2023, 5 abril). https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

-Corrales, M. (s. f.). *ALGORITMO SCAN O ALGORITMO DEL ASCENSOR*. prezi.com. https://prezi.com/p/tvisfzk3grnb/algoritmo-scan-o-algoritmo-del-ascensor/

- OpenAI. (2023). ChatGPT (Modelo de lenguaje grande) [Software]. Recuperado de https://openai.com/chatgpt

-Presentaciones vistas en clase

-Microsoft. (2023) Copilot (Modelo de lenguaje grande) [Software].

-Domingo, A. (2020, 21 septiembre). *¿Cómo funciona un ascensor y qué tipos de elementos lo integran? - Ascensores Domingo*. Ascensores Domingo. https://ascensoresdomingo.com/como-funciona-un-ascensor/