Diseño y Programación Orientados a Objetos

1er. Semestre 2025

Tarea 1: Simulando el Patrón de Diseño Publicador y Suscriptor

Lea detenidamente la tarea. **Si algo no lo entiende, consulte. Si es preciso, se incorporarán aclaraciones al final**. Esta interacción se asemeja a la interacción entre desarrolladores y clientes con el fin de aclarar requerimientos no del todo especificados.

1. Objetivos de la tarea

- Modelar objetos reales como objetos de software.
- Ejercitar la creación y extensión de clases dadas para satisfacer nuevos requerimientos.
- Reconocer clases y relaciones entre ellas en código fuente Java.
- Ejercitar la compilación y ejecución de programas en lenguaje Java desde una consola de comandos.
- Ejercitar la configuración de un ambiente de trabajo para desarrollar aplicaciones en lenguaje Java, se puede trabajar con un editor tipo "Sublime" o con un IDE. IntelliJ es el IDE sugerido tanto para esta tarea como la Tarea 2.
- Ejercitar la entrada y salida de datos en Java.
- Manejar proyectos vía GIT (voluntario para esta tarea).
- Conocer el formato .csv y su importación hacia una planilla electrónica.
- Ejercitar la preparación y entrega de resultados de software (creación de makefile, readme, documentación).
- Familiarización con desarrollos "iterativos" e "incrementales" (o crecientes).

2. Descripción General

Esta tarea busca practicar la orientación a objeto en un sistema que simula la operación del patrón publicador y suscriptor (Figura 1). Este patrón será usado para anunciar eventos de forma asincrónica a varios consumidores interesados, sin necesidad de vincular directamente generadores de eventos con sus receptores.

Se usará este patrón para enviar mensajes a un canal o tópico, el cual será recibido por quienes suscriban ese canal o tópico. Así, instancias de publicadores podrán enviar mensajes a un tópico o canal, los cuales serán recibidos por instancias de suscriptores al mismo tópico o canal. Con la misma arquitectura (el mismo patrón), instancias de publicadores podrán enviar mediciones los cuales serán recibidos por suscriptores al canal temperatura o posición, por ejemplo. Se podría simular así un sistema IoT (Internet of Things) de reporte temperatura (de una máquina, por ejemplo) o posición (de un automóvil).

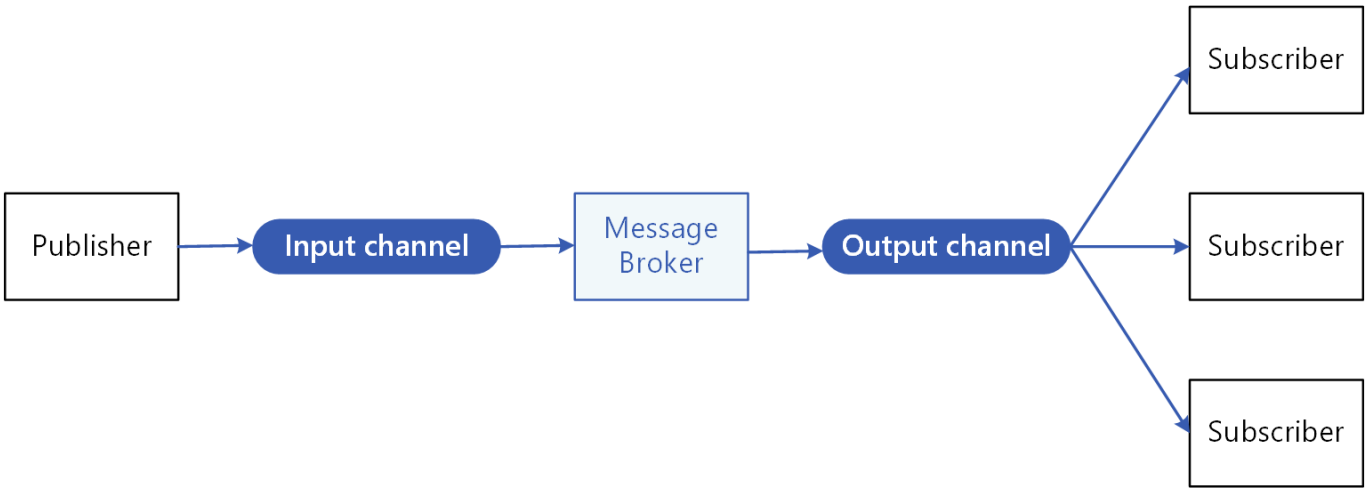

Figura 1: Componentes lógicos del patrón publicador y suscriptor (fuente).

El patrón general de la Figura 1 es usado para simular dos tipos de publicador-suscriptor, (Figura 2). El primero conecta streamers con su comunidad a través de notificaciones. Si bien les llamaremos streamers, estos solo transmitirán mensajes de una línea de texto (anuncios, saludos, recordatorios). Para cada streamer puede haber cero o varios seguidores los cuales reciben actualizaciones de las notificaciones publicadas y las escriben en un archivo de salida. La idea de este par es simular una versión simplificada (solo texto) de plataformas de streaming reales como Twitch y Kick.

El diseño base de este patrón también será usado para simular la interacción entre el GPS de móviles y suscriptores a sus posiciones. En esta simulación, el GPS de un móvil lee sus posiciones ingresadas por el usuario vía el teclado y este publicador las sube al bróker para su difusión. Dos tipos de suscriptores existen para las posiciones de un móvil: un Registrador de posiciones y un Monitor de posición. Un registrador almacena las actualizaciones de la posición en un archivo de texto con formato CSV (Comma-separated values). Esto permite que usted, desde una planilla Excel, importe este archivo y luego haga un gráfico que muestre la trayectoria del móvil. Naturalmente, esta manipulación debe ser hecha fuera del programa. Un Monitor de posición reporta la posición sólo cuando distancia es superior a 500 desde el origen $(0,0)$. Las posiciones están dentro de un plano de coordenada x e y con $x, y \in [0, 500]$.

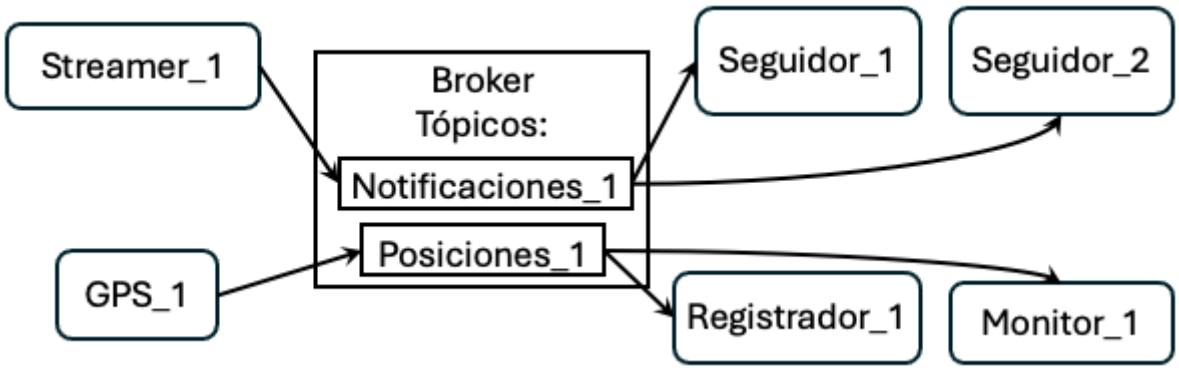

Figura 2: Simulación de difusión de notificaciones y posiciones usando patrón publicador-suscriptor *(agregar imagen aquí)*.

En Figura 2 se muestra un streamer y un GPS instancias de publicadores de notificaciones y posiciones respectivamente. Ambos envían líneas de texto ingresadas por teclado y dirigidas a distintos tópicos. Del lado derecho tenemos dos seguidores suscritos al tópico "Notificaciones_1" y dos suscriptores al tópico "Posiciones_1".

2.1 Funcionamiento de la aplicación

La simulación es configurada por un archivo de entrada (por ejemplo `config.txt`) pasado como argumento del programa. Este archivo de configuración tiene el siguiente formato según si especificamos un publicador o un suscriptor (la notación "|" significa que cada línea sigue el formato de la izquierda o la derecha):

```
(publicador <nombre> <tópico>)|(suscriptor <tipo> <nombre> <tópico> <archivo>)
```

- `<nombre>`: Este campo usado para distinguir un publicador o suscriptor de otro y eventualmente hacer más descriptivo al mostrar su estado. Si el nombre de un suscriptor está en más de una línea, significa que desea suscribirse a más de un tópico. Un publicador sólo publica en un tópico.
- `<tópico>`: es un string que representa el canal o tópico de recepción de publicaciones a las cuales los suscriptores están suscritos.
- `<tipo>`: representa el tipo de suscriptor. En esta tarea hay 3 tipos de suscriptores con distintos comportamientos y por deben ser distinguidos. Éstos son: `Seguidor`, `Registrador` y `Monitor`. Seguidores sólo se suscriben a los tópicos donde publican Streamers. Registradores y Monitores se suscriben a tópicos asociados a GPSs.
- `<archivo>`: es el nombre del archivo donde un suscriptor escribe las salidas generadas luego de recibir los eventos de su tópico.

Ejemplo archivo configuración:

```
publicador Streamer_1 Notificaciones_1
suscriptor Seguidor Seguidor_1 Notificaciones_1 seguidor_1.txt
suscriptor Seguidor Seguidor_2 Notificaciones_1 seguidor_2.txt
publicador GPS_1 Posiciones_1
suscriptor Registrador Registrador_1 Posiciones_1 ruta_1.txt
suscriptor Monitor Monitor_1 Posiciones_1 ubicacion_1.txt
```

Formato eventos (entrada teclado): Los eventos de esta simulación son ingresados por el usuario vía el teclado. El formato de los eventos es (un evento por línea):

```
<nombre del publicador> <mensaje>
```

Ejemplo:

```
Streamer_1  ¡Nuevo stream este viernes a las 19:00!
GPS_1   20  150
Streamer_1  Hoy revisaremos estructuras de datos
GPS_1   20  20
GPS_1   100 20
GPS_1   100 150
GPS_1   200 150
```

En general el mensaje publicado por Streamer y GPS corresponde al texto luego de su nombre.

Ejecución:

La ejecución del programa será del tipo:

```
$ java Simulador config.txt
```

Donde `config.txt` contiene la configuración del simulador.

Formato salida Seguidor:

La salida de un seguidor tiene el siguiente formato (una notificación por línea):

```
<nombre> <tópico> <notificación>
```

Ejemplo:

```
Seguidor_1  Notificaciones_1    ¡Nuevo stream este viernes a las 19:00!
Seguidor_1  Notificaciones_2    Hoy revisaremos estructuras de datos
```

Formato salida Registrador (CSV):

La salida de un registrador tiene el siguiente formato (una posición por línea):

```
<nombre>,<tópico>,<posición_x>,<posición_y>
```

Ejemplo

```
Registrador_1,  Posiciones_1,   20, 150
Registrador_1,  Posiciones_1, 20,   20
.
.
```

Para la salida del registrador se pide usar el formato CSV de manera que luego usted pueda importar los datos a una planilla Excel y pueda en ella hacer un gráfico con la trayectoria.

3. Desarrollo en Etapas

Para llegar al resultado final de esta tarea usted debe aplicar una metodología de desarrollo "Iterativo y creciente (o incremental)" para desarrollo de software. Usted y su equipo irán desarrollando etapas donde los requerimientos del sistema final son abordados gradualmente. En cada etapa usted y su equipo obtendrá

una solución que funciona para un subconjunto de requerimientos finales o bien es un avance hacia ellos. En AULA se dispondrá el recurso para subir la solución correspondiente a cada etapa del desarrollo. **Su equipo deberá entregar una solución para cada una de las etapas aun cuando la última integre las primeras. El readme y archivo de documentación deben ser preparados sólo para la última etapa. Prepare y entregue un makefile para cada una de las etapas**. Esto tiene por finalidad, educar en la metodología de desarrollo iterativo e incremental.

3.1 Primera Etapa: Un Streamer envía notificaciones que almacena un Seguidor

En esta etapa se deben crear, al menos, las clases `Publisher`, `Subscriber`, `Follower` y `Broker`. Un objeto streamer y más adelante un objeto GPS pueden ser considerados instancias de `Publisher`. La clase `Subscriber` modela el funcionamiento genérico de un suscriptor. La clase `Follower` especializa al tipo de suscriptor que sigue las notificaciones de un streamer. Como los publicadores y suscriptores tienen atributos comunes (al menos su nombre), tiene sentido definir la clase `Component` (componente) como padre de estas dos. `Broker` debe administrar los tópicos. En esta etapa la configuración estará definida en el `main` de la clase `T1Stage1` y sin requerir la lectura desde un archivo externo. Aquí hay un código ejemplo para la clase `T1Stage1`. La entrada será ingresada por teclado, pero como existe sólo un publicador, se puede omitir su nombre en cada línea y sólo poner el mensaje de la notificación. El archivo de salida de esta etapa tendrá el formato definido para un seguidor y sus mensajes coincidirán con las líneas de texto ingresadas por teclado. Para observar la salida, luego de iniciar el programa, en otra consola bash, ejecute el comando:

```
$ tail -f seguidor.txt
```

Nota: puede ser de utilidad este diagrama de clases.

3.2 Segunda Etapa: GPS publica posiciones que almacena un Registrador

Se pide crear la clase `Recorder` la cual es similar a la clase `Follower`, pero su salida es grabada en formato CSV según se indica al final de la sección 2.1. Instancias de la clase Publisher deberían funcionar como GPS enviando string de texto leídos desde el teclado y con las coordenadas x e y de cada posición reportada. Use el mismo formato indicado en sección 2.1, excepto que por tratarse de un único publicador no necesita ingresar su nombre al comienzo de cada línea. La configuración de la simulación en esta etapa será a través de un archivo de configuración en el cual se definirá un publicador y luego un suscriptor instancia de Recorder. Aquí hay un archivo de configuración posible. Para observar la salida, luego de iniciar el programa, en otra consola bash, ejecute el comando:

```
$ tail -f trayectoria.txt
```

Nota: puede ser de utilidad este diagrama de clases.

3.3 Tercera Etapa: Streamer envía notificaciones a dos suscriptores

Generalice el método `setupSimulation` del programa previo para configurar un simulador que permita configurar un streamer y dos seguidores de sus publicaciones. La entrada seguirá siendo por teclado, pero esta vez no omita el nombre del streamer. Si se ingresa un nombre de streamer errado, el programa debe

desplegar "Unknown Publisher" y no considerar su mensaje. Usando el comando tail en otras dos consolas, usted podrá ver la salida de cada seguidor.

3.4 Cuarta Etapa: Número arbitrario de publicadores y suscriptores incluyendo monitores

Cree la clase `Monitor`, la cual es semejante a `Recorder`, excepto que cambia la forma de generar las salidas a archivo. Se espera que el archivo de configuración pueda incluir varios publicadores y varios tipos de suscriptores. En esta etapa llame Simulador su programa ejecutable.

3.5 Extra-crédito

Esta parte es voluntaria. Su desarrollo otorga **5 puntos adicionales**. Si desarrolla esta parte, indíquelo en su documentación. Cree un nuevo tipo de suscriptor el cual muestra por pantalla el número total de mensajes publicados en los tópicos que está suscrito. El formato de la salida es:

```
<tiempo en segundos desde que partió el programa>, <número de mensajes
acumulados>
```

Luego de importar estos datos en Excel, genere un gráfico número de mensajes versus tiempo e inclúyalo en su documentación.

Adicionalmente, se otorgarán **3 puntos adicionales** a aquellos grupos que entreguen su tarea utilizando la plataforma Github.

En ambos casos, **la nota final se satura en 100**.

4. Elementos de su documentación

Entregue todo lo indicado en "Normas de Entrega de Tareas". Prepare un archivo makefile para compilar y ejecutar su tarea en aragorn con rótulo “run”. Además, incluya rótulos "clean" para borrar todos los .class generados. Los comandos usados en cada caso son:

```
$ make         # Compilar
$ make run     # Ejecutar
$ make clean   # Limpiar archivos .class
```

En su archivo de documentación (pdf o html) incorpore el diagrama de clases (UML) de la aplicación (etapa 4).

Ayuda

1. Para mostrar en una consola el contenido de un archivo de texto en la medida que este está siendo escrito, en Linux (compatible con Aragorn), usted puede usar el comando tail, en particular la opción -f. Así, al ejecutar:

```
$ tail -f salida.txt
```

en pantalla usted verá el contenido del archivo salida.txt en la medida que este crece en contenido. 2. En Linux es posible redireccionar a un archivo la salida que normalmente va a pantalla.

```
$ java Simulador > salida.txt
```

Almacenará en `salida.txt` todo lo que el programa envíe a la salida estándar. 3. Si desea acelerar pruebas de su programa, una vez creados los archivos de salida (por ejemplo manualmente), usted puede ejecutar su programa redirigiendo la entrada de datos:

```
$ java Simulador < eventos.txt
```

4. Para los extra-créditos, revise el programa `TimeGoesByTest.java`.