A2

Diseño y procesamiento de datos de un sistema de log

INFORME

PARTICIPANTE: PABLO CÉSAR GARZÓN BENÍTEZ

En este documento se registra el desarrollo de la actividad de diseño y procesamiento de datos de un sistema de log aplicando los principios de diseño de software y POO. Para ello se considera la construcción de un sistema de procesamiento para eventos de interacción de usuarios de una nueva aplicación móvil.

El flujo de actividades realizados por el usuario final y la captura de datos se observa visualmente a el siguiente diagrama de caso de uso (ver figura 1):

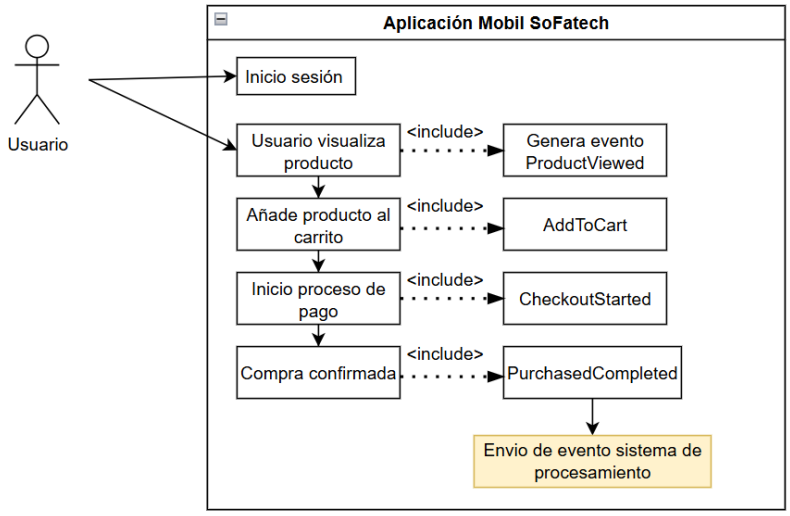


Figura 1. Caso de Uso. Flujo de actividades compra

Teniendo en cuenta el esquema general de pasos para la realización de una compra se consideran los siguientes aspectos para el procesamiento de los datos del sistema.

1. MODELADO DE DATOS

Se debe considerar que al usuario ingresar a la aplicación se registran una serie de datos potencialmente descriptores de las situaciones que ocurren en cada compra, al realizar el registro de cada uno de los pasos mostrados en el caso de uso anterior es importante considerar que los eventos deben contar con atributos como:

- Tipo de evento
- Fecha y hora del evento
- Identificador de usuario que activo el evento
- Información dispositiva desde el que se ingreso
- Detalles de la compra (producto y cantidad)
- Versión de la aplicación móvil (Opcional)

Ahora, considerando los eventos de ver producto y añadir carrito, estos consideran campos de información con algunas diferencias en los atributos de interés o de información que proveen, por ejemplo, se podría establecer una plantilla para el evento tal que tengamos 3 elementos clave:

1. Datos generales del evento
2. Propiedades del evento
3. Coordenadas geográficas del evento (opcional)

El único cambio observable se encontraría en el paso 2, donde para el caso de ‘agregar_carrito’ nos interesa tener información sobre el id_producto, cantidad, precio_unitario, id_cart, mientras que

para el evento ‘ver_producto’ las propiedades de interés es reconocer nombre_producto, id_producto, categoría y ruta_referencia. En definitiva para todos los eventos los campos similares son los datos de registro de log respecto al identificador de log, fecha de registro de la acción, identificador del usuario de la acción e información del dispositivo de conexión a la aplicación si es necesario, mientras que los campos particulares de cada evento deben considerar el propósito principal del evento tal como información relacionada a ver una compra, agregar una compra, registro de credenciales de compra y confirmación de la compra.

Los campos que se definen en la siguiente tabla establecen las propiedades que definitivamente deben tener todos los logs.

Campos obligatorios logs	**Tipo dato**	**Campos opcionales**	**Tipo dato**
Tipo_Evento	String	Localización	Struct/object
Fecha_accion	timestamp	Cart_id	String
User_id	String	Estado_evento	String
Info_dispositivo	Struct/Object	-	-
Propiedades_evento	Struct/Object	-	-

Ahora de forma práctica la estructura de un log puede ser legible y visualmente visto mediante estructuras como JSON, pero para manejarse en nuestro sistema de datos es aconsejable utilizar un esquema tipo AVRO (Procesamiento de eventos a gran escala) o Parquet (Procesamiento para analítica o almacenar en DWH). El esquema conceptual de los datos que se tendrían serían los siguientes:

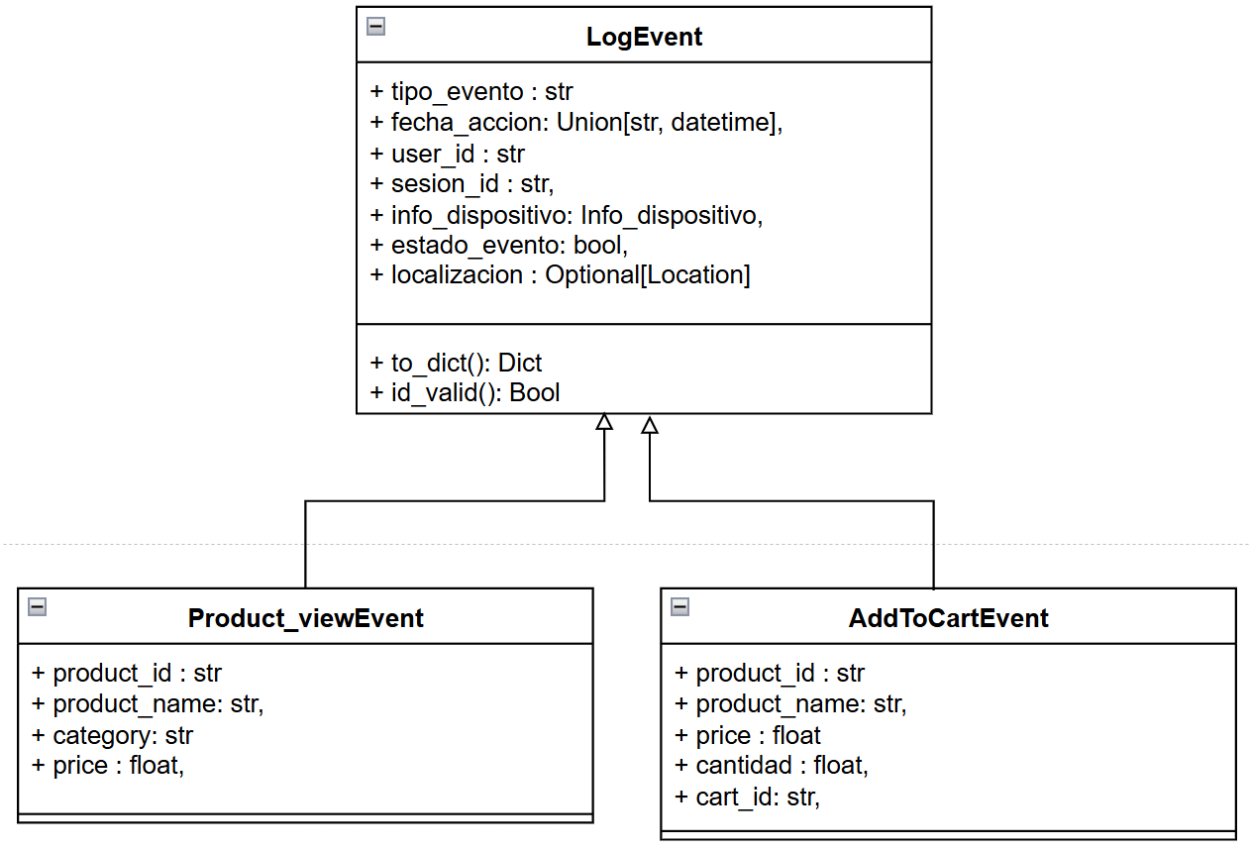
[Sección principal] Tipo_evento : String Fecha_evento : Timestamp User_id : String Info_dispositivo: Struct Os: String Model: String App_version: String [Sección secundaria] – Depende del evento	**EJEMPLO:** Tipo_evento : “Vista producto” Fecha_evento : “2025-06-26-18:02:15” User_id : “U325452” Info_dispositivo: Struct Os: “Symbian” Model: “Nokia sky X32” App_version: “2.4.0” [Sección secundaria] – Depende del evento

Para vista producto	**Para vista producto**
Id_producto: String	Id_producto: “R436”
Nombre_producto: String	Nombre_producto: “Teclado Gamer”
Categoria: String	Categoria: “Electrónica”
Precio_unitario: Double	Precio_unitario: 3002.33
Para adicionar carrito producto	**Para adicionar carrito producto**
Precio_unitario: Double	Precio_unitario: 3002.33
Precio: Double	Precio_total: 12009.20
Cantidad: Int	Cantidad: 4
Card_id : String (Opcional)	Card_id : “C20250629-U10293”

2. DISEÑO ORIENTADO A OBJETOS Y PATRONES

A continuación, se utiliza el paradigma orientado a objetos y los patrones de diseño creacional y comportamental Factory y chain of responsability para representar los eventos de la aplicación.

Utilizando la programación orientada a objetos se construye la clase base llamada LogEvent, que tendrá un método para la validación de datos y el método to_dict.



Aplicábamos el principio de Open/Closed Principle (OCP) para heredar los atributos de la clase padre LogEvent permitiendo la extensión de las funcionalidades de una clase base sin hacer modificaciones críticas, adicionalmente se crean funcionalidades individuales para los tipos de logs como Product_view_Event y AddToCartEvent. Para conocer el desarrollo de estas clases dirigirse a: detalles_log.py

Posteriormente crearmos la clase EventFactory para estructurar los datos en crudo que lleguen de la aplicación móvil y devolver instancias de ProductViewEvent, AddToCartEvent o LogEvent genérico si no reconoce el tipo. La clase EventFactory emplea el patrón de diseño Factory para alterar la creación de objetos según subclases y la alineación de los datos con determinadas propiedades. Dependiendo si el evento es de tipo **ProductViewed** o **AddTocart** el objeto creado con la clase EventFactory permitirá instanciar diferentes tipos de objetos desde un mismo punto centralizado. El desarrollo de este módulo se encuentra implementado en: FactoryPatron.py

El patrón Factory contiene diferentes ventajas tales como desacoplar la lógica de creación de objetos facilitando el mantenimiento y la extensión del código, hacer escalables los requisitos de los eventos al solo modificar la fábrica, no el resto del sistema y finalmente la legibilidad del código, este puede ser más limpio y fácil de entender, ya que delega la creación de objetos a un tercero.

Ahora para la implementación de una cadena de validadores mediante el uso de una clase base llamada BaseValidador con métodos para conectar los handlers o manejadores tal como validate y el puntero next_validator se realizan comprobaciones respecto a evaluar que en los Logs no existan campos vacíos y que el tipo de evento de los eventos sea obligatorio y corresponda a la lista de eventos admitidos. En el proceso de construcción se considera el principio SRP (Single Reponsibility) para obtener una única funcionalidad por función. Los validadores se encadenan dinámicamente y cuando se encuentre algún error se etiqueta al evento como invalido. Se manejan los errores mediante la utilización de mensajes específico para carencias en los campos de información registrando los múltiples errores para ser analizados a posterior. chain_of_responsability.py

El principio de Responsabilidad única en esta situación (SRP) permite organizar los validadores como clases independientes, en el caso que exista un error en alguno de estos validadores podremos acceder directamente a hacer los cambios sobre la parte el código afectado sin alterar el comportamiento de partes críticas de los diferentes módulos. Por otro lado, el patrón de responsabilidad es capaz de desacoplar las reglas, pero encadenar las validaciones por un proceso de eventos secuenciales. Este patrón hace que la lógica se mantenga modular y extensible permitiendo agregar, guitar y reordenar validadores sin modificar el resto del código.

Para la simulación del procesamiento de log creamos una función para la generación de eventos de forma sintética con la función y así poner a prueba la cadena de responsabilidad mediante un ciclo iterativo, la función de generación de eventos se puede encontrar aquí: generador_datos_sinteticos.py, esta función agrega pequeños errores en algunos de los eventos para hacer mantener un equilibrio entre eventos validos e inválidos. Finalmente, desde una función principal llamada main.py se imprime en consola la creación, validación y agregación de todos los eventos a un vector de resultado para imprimir en pantalla.

Modelando en un diagrama de secuencias el proceso por el que pasa un evento log desde su creación hasta su calificación se observa a continuación:

Secuencia de pasos:

1. **main** llama a generar_eventos_prueba(15) en **generador_datos_sinteticos** → retorna lista de eventos de prueba.
2. **main** crea una instancia de **EventValidatorChain**.
3. **main** agrega validadores (NotNullValidator, EventTypeValidator, ProductDetailsValidator) a la cadena.
4. **main** llama a build_chain() en **EventValidatorChain**.
5. Para cada evento en la lista:
    - **main** llama a EventFactory.crear_evento(evento_bruto) → retorna un objeto evento.
    - **main** llama a validator_chain.validate(evento):
        - **EventValidatorChain** pasa el evento por cada validador en orden.
        - Cada validador puede agregar errores a la lista si corresponde.
    - **main** guarda el resultado (incluyendo el dict del evento y errores).
    - **main** imprime resumen del evento.
6. Al final, **main** imprime el resumen de validación y detalles de eventos inválidos.

Para consultar el código fuente del proyecto dirigirse:
https://github.com/PabloGarzonB0/procesamiento_logs_app_movil

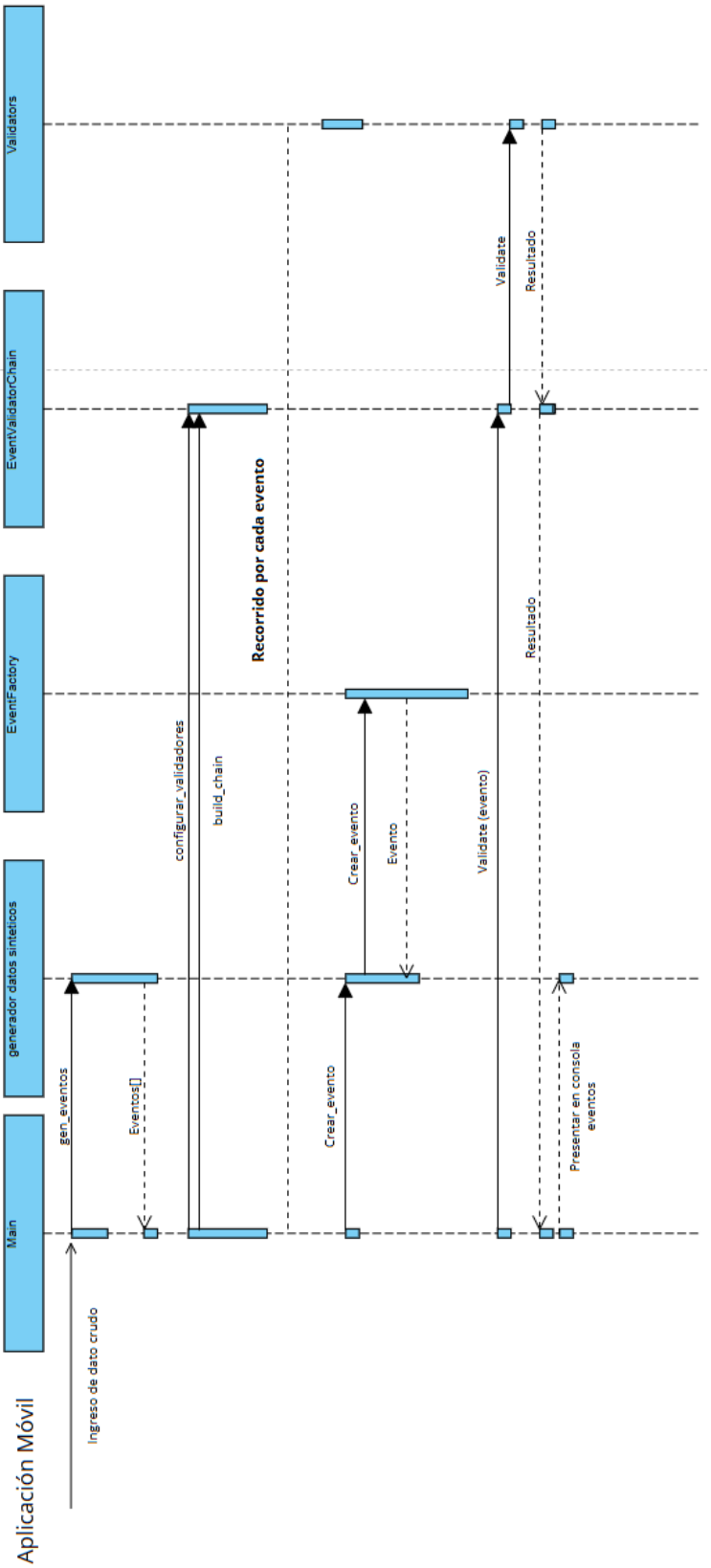


Figura 2. Diagrama de secuencia del flujo de procesamiento de un solo evento de log

En el caso que se tubieran que almacenar estos evento en una base de datos SQL o NoSQL se deberia crear una tabla principal para los eventos, conciderando los campos comunes (id, tipo_evento, fecha_accion, user_id, sesion_id, etc). Los atributos específicos de cada tipo de evento pueden ser almacenarse en tablas hijas empleadas como herencia tal como product_view_events y add_to_cart_events, cada una con su clave foránea a la tabla principal y sus columans específicas.

```sql
CREATE TABLE product_view_events (
    evento_id INT REFERENCES eventos(id),
    product_id VARCHAR(50),
    product_name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL,
    PRIMARY KEY (evento_id)
);
```

```sql
CREATE TABLE eventos (
    id SERIAL PRIMARY KEY,
    tipo_evento VARCHAR(50),
    fecha_accion TIMESTAMP,
    user_id VARCHAR(50),
    sesion_id VARCHAR(50),
    -- otros campos comunes
);
```

```sql
CREATE TABLE add_to_cart_events (
    evento_id INT REFERENCES eventos(id),
    product_id VARCHAR(50),
    product_name VARCHAR(100),
    price DECIMAL,
    cantidad INT,
    cart_id VARCHAR(50),
    PRIMARY KEY (evento_id)
);
```

En el caso de tener una base de datos NoSQL (documento) cada evento se almacena como tipo JSON, se incluyen una serie de datos comunes y específicos en el documento y se pueden guardar en una base de datos tipo MongoDB. La estructura de los datos almacenados puede ser la siguiente:

```json
{
  "tipo_evento": "AddTocart",
  "fecha_accion": "2025-06-30T12:34:56",
  "user_id": "u123",
  "sesion_id": "s456",
  "product_id": "p789",
  "product_name": "Zapatos",
  "price": 99.99,
  "cantidad": 2,
  "cart_id": "c001",
  "info_dispositivo": {
    "os": "Android",
    "model": "Samsung S21",
    "app_version": "1.2.3"
  }
}
```