TAREA #3 - PRUEBAS Y REFACTORIZACIÓN

GRUPO#3

- CABRERA YAGUAL ELIAS EMANUEL
- CLAUDETT VALERO JOEL JOSUE
- LUIS GUZMAN
- GARCIA ERAZO ROY ELIECER

Contenido

SECCIÓN C: Code Smells

Code Smell 1: Duplicate Code

Descripción del problema:
Hay duplicación de código en las clases encargadas de notificar los eventos del sistema, donde se repite la misma lógica de impresión de mensajes, variando únicamente el destinatario.

Impacto en el sistema:
La duplicación incrementa el esfuerzo de mantenimiento, ya que cualquier cambio en el formato del mensaje debe hacerse en varias clases, lo cual aumenta el riesgo de inconsistencias.

Ubicación:

Técnica de refactorización sugerida:

- Extract Method para centralizar la lógica común.
- Pull Up Method mediante una clase base compartida.

Code Smell 2: Long Method

Descripción del problema:
Existen varios métodos que concentran distintas responsabilidades dentro de un mismo bloque de código, como validaciones, cambios de estado, creación de objetos y notificaciones.

Impacto en el sistema:
Los métodos largos reducen la legibilidad y dificultan la comprensión del flujo lógico, además de complicar las pruebas unitarias y el mantenimiento.

Ubicación:

- Método confirmarCompra en la clase función

```java
public Compra confirmarCompra(Usuario usuario, List<String> codigosAsiento) {
    Objects.requireNonNull(usuario);
    Objects.requireNonNull(codigosAsiento);

    liberarReservasVencidas();

    List<Asiento> comprados = new ArrayList<>();
    for (String codigo : codigosAsiento) {
        Asiento a = buscarAsiento(codigo);
        if (a.getEstado() == EstadoAsiento.AGOTADO) {
            throw new IllegalStateException("El asiento " + codigo + " ya esta agotado");
        }
        if (a.getEstado() == EstadoAsiento.DISPONIBLE) {
            throw new IllegalStateException("El asiento " + codigo + " no esta reservado");
        }
        a.confirmarCompra();
        comprados.add(a);
    }

    Compra compra = new Compra(usuario, nombreEspectaculo, fechaHora, comprados);

    notificaciones.avisar(new EventoFuncion(
            tipo: "Compra",
            "Compra confirmada para " + usuario.getNombre() + " (" + comprados.size() + " asientos)",
            nombreEspectaculo,
            fechaHora
    ));

    return compra;
}
```

- Método reservar en la clase Funcion

Técnica de refactorización sugerida:

- Extract Method para separar responsabilidades.
- Decompose Conditional para clarificar las reglas de negocio.

Code Smell 3: Large Class

Descripción del problema:
La clase Funcion agrupa demasiadas responsabilidades relacionadas con la gestión de asientos, reservas, compras y notificaciones. Responsabilidades que no están relacionadas con la clase.

Impacto en el sistema:
Una clase de gran tamaño presenta baja cohesión y se convierte en un punto crítico de cambios, afectando la estabilidad general del sistema.

Ubicación:

- Clase Funcion

Técnica de refactorización sugerida:

- Extract Class para dividir responsabilidades en clases especializadas.
- Move Method para redistribuir lógica según el dominio.

Code Smell 4: Primitive Obsession

Descripción del problema:
Se utilizan tipos primitivos y cadenas de texto para representar conceptos del dominio que poseen reglas propias, como códigos de asientos o tipos de evento.

Impacto en el sistema:
Este enfoque incrementa la posibilidad de errores y obliga a repetir validaciones en múltiples puntos del código.

Ubicación:

- Uso de String para códigos de asiento

```
private final Seccion seccion;
private final int fila;
private final int numero;
private final double precio;
```

- Uso de String para el tipo de evento

```
private final String tipo;
private final String mensaje;
private final String nombreEspectaculo;
private final LocalDateTime fechaHora;
```

- Uso de String como identificador de espectáculos

```
private final String nombre;
private final List<Funcion> funciones;
```

Técnica de refactorización sugerida:

- Replace Primitive with Object para encapsular reglas del dominio.

Code Smell 5: Long Parameter List

Descripción del problema:
Se detectaron constructores y métodos que reciben un número elevado de parámetros relacionados.

Impacto en el sistema:
Las llamadas se vuelven propensas a errores y disminuye la claridad del código, especialmente al reutilizar dichos métodos.

Ubicación:

- Constructor de Función

- Constructor de Compra

- Método crearFuncion en EnVivoTicketsFacade

Técnica de refactorización sugerida:

- Introduce Parameter Object para agrupar datos relacionados.

Code Smell 6: Inappropriate Intimacy

Descripción del problema:
La capa de dominio depende directamente de una clase concreta de la capa de servicios para la notificación de eventos.

Impacto en el sistema:
Este acoplamiento reduce la independencia del dominio y dificulta la evolución del sistema o la realización de pruebas unitarias.

Ubicación:

- Clase Funcion dependiendo de CentroNotificacionesFuncion

Técnica de refactorización sugerida:

- Replace Dependency with Abstraction, utilizando interfaces.

Code Smell 7: Divergent Change

Descripción del problema:
Una misma clase puede cambiar por múltiples razones diferentes, relacionadas con reglas de reservas, compras o notificaciones.

Impacto en el sistema:
Esto incrementa la probabilidad de errores al modificar el código, ya que cambios no relacionados afectan a la misma clase.

Ubicación:

- Clase Funcion

Técnica de refactorización sugerida:

- Extract Class para asegurar una sola razón de cambio.

Code Smell 8: Shotgun Surgery

Descripción del problema:
Un cambio pequeño en el manejo de eventos o mensajes requiere modificar varias clases diferentes.

Impacto en el sistema:
El mantenimiento se vuelve disperso y aumenta el riesgo de olvidar aplicar un cambio en algún punto del sistema.

Ubicación:

- Funcion
- EnVivoTicketsFacade
- Clases notificadoras

Técnica de refactorización sugerida:

- Extract Class para centralizar la gestión de eventos.

Code Smell 9: Data Class

Descripción del problema:
Algunas clases contienen únicamente atributos y métodos de acceso, sin comportamiento propio.

Impacto en el sistema:
La lógica relacionada queda dispersa en otras clases, debilitando el modelo orientado a objetos.

Ubicación:

- Clase EventoFuncion
- Clase Usuario

Técnica de refactorización sugerida:

- Move Method para incorporar comportamiento al objeto.

Code Smell 10: Middle Man

Descripción del problema:
La clase fachada reenvía llamadas directamente a otras clases sin aportar lógica adicional en ciertos métodos.

Impacto en el sistema:
Se incrementa la complejidad estructural sin un beneficio claro en esos casos.

Ubicación:

- Clase EnVivoTicketsFacade

Técnica de refactorización sugerida:

- Move Method cuando la fachada no agrega valor real.
- Mantener la fachada solo como punto de entrada del sistema.

SECCIÓN D

Refactorización: Duplicate Code (Lógica de Notificación)

- Code Smell Corregido: Duplicate Code en Funcion.java.
- Técnica Aplicada: *Extract Method*.
- Justificación: La creación del objeto EventoFuncion y la llamada a notificaciones.avisar se repetía en cambiarFecha, reservar y confirmarCompra. Al extraerlo a un método privado, centralizamos la lógica de notificación, reduciendo errores si el formato del evento cambia.
- Verificación: Las pruebas unitarias de notificación siguen pasando, confirmando que los mensajes se envían correctamente.

Antes

```
// En cambiarFecha

notificaciones.avisar(new EventoFuncion("Reprogramacion", "La funcion ", nombreEspectaculo, fechaHora));


// En reservar

notificaciones.avisar(new EventoFuncion("Reserva", "Se reservaron", nombreEspectaculo, fechaHora));


// En confirmarCompra

notificaciones.avisar(new EventoFuncion("Compra", "Compra confirmada", nombreEspectaculo, fechaHora));
```

Despues

```
// Método extraído

private void notificarEvento(String tipo, String mensaje) {

   notificaciones.avisar(new EventoFuncion(tipo, mensaje, nombreEspectaculo, fechaHora));

}


public void cambiarFecha(LocalDateTime nuevaFechaHora) {

   fechaHora = Objects.requireNonNull(nuevaFechaHora);

   notificarEvento("Reprogramacion", "La funcion cambio de fecha/hora");
```

}

Refactorización: Long Method (Validación de Asientos)

- **Code Smell Corregido:** Long Method en Funcion.reservar y Funcion.confirmarCompra.
- **Técnica Aplicada:** *Extract Method*.
- **Justificación:** Los métodos originales mezclaban la búsqueda de asientos, la validación de disponibilidad y la lógica de negocio. Extraer la validación mejora la legibilidad y permite reutilizar la lógica de búsqueda segura.
- **Verificación:** Las pruebas de excepciones (IllegalStateException) pasan correctamente al intentar reservar asientos ocupados.

Código antes

```
public List<Asiento> reservar(List<String> codigosAsiento, Duration duracion) {
   liberarReservasVencidas();
   List<Asiento> seleccion = new ArrayList<>();
   for (String codigo : codigosAsiento) {
      Asiento a = buscarAsiento(codigo);
      if (a.getEstado() != EstadoAsiento.DISPONIBLE) {
         throw new IllegalStateException("El asiento " + codigo + " no esta disponible");
      }
      seleccion.add(a);
   }
}
```

código después

```
public List<Asiento> reservar(List<String> codigosAsiento, Duration duracion) {
   validarDuracion(duracion); // Otro método extraído
   liberarReservasVencidas();
   List<Asiento> seleccion = obtenerAsientosParaReserva(codigosAsiento);

   return seleccion;
}
```

```
private List<Asiento> obtenerAsientosParaReserva(List<String> codigos) {
   List<Asiento> seleccion = new ArrayList<>();
   for (String codigo : codigos) {
      Asiento a = buscarAsiento(codigo);
      if (a.getEstado() != EstadoAsiento.DISPONIBLE) {
         throw new IllegalStateException("El asiento " + codigo + " no esta disponible");
      }
      seleccion.add(a);
   }
   return seleccion;
}
```

Refactorización: Primitive Obsession (Tipos de Espectáculo)

- **Code Smell Corregido:** Primitive Obsession en Espectaculo.java y sus subclases.
- **Técnica Aplicada:** *Replace Type Code with Class/Enum*.
- **Justificación:** El método getTipo() devolvía un String ("Teatro", "StandUp"), lo cual es propenso a errores de tipeo (ej. "teatro" vs "Teatro"). Se crea un Enum TipoEspectaculo para seguridad de tipos.

Antes

```
public abstract String getTipo();


// En Teatro.java
public String getTipo() { return "Teatro"; }
```

Despues

```
// Nuevo Enum
public enum TipoEspectaculo {
   TEATRO, STAND_UP, MICROTEATRO
}


// En Espectaculo.java
public abstract TipoEspectaculo getTipo();
```

```
// En Teatro.java
@Override
public TipoEspectaculo getTipo() {
   return TipoEspectaculo.TEATRO;
}
```

Refactorización: Inappropriate Intimacy (Dependencia Concreta)

- **Code Smell Corregido:** Inappropriate Intimacy en Funcion.java.
- **Técnica Aplicada:** *Replace Dependency with Abstraction (Extract Interface)*.
- **Justificación:** Funcion dependía directamente de la clase concreta CentroNotificacionesFuncion. Esto hace difícil testear Funcion sin enviar notificaciones reales. Se extrae la interfaz ServicioNotificacion para desacoplar el dominio del servicio.

Antes

```
private final CentroNotificacionesFuncion notificaciones;

public Funcion(..., CentroNotificacionesFuncion notificaciones) {
   this.notificaciones = notificaciones;
}
```

Después

```
// Nueva Interfaz
public interface ServicioNotificacion {
   void avisar(EventoFuncion evento);
}

// En Funcion.java
private final ServicioNotificacion notificaciones;

public Funcion(..., ServicioNotificacion notificaciones) {
   this.notificaciones = notificaciones;
}
```

Refactorización: Long Parameter List (Constructor de Compra)

- **Code Smell Corregido:** Long Parameter List en Compra.java.
- **Técnica Aplicada:** *Introduce Parameter Object*.
- **Justificación:** El constructor de Compra recibía muchos datos sueltos (String nombreEspectaculo, LocalDateTime fecha). Estos datos pertenecen cohesivamente a la información de la función. Se crea un objeto InfoFuncion (o se pasa el objeto Funcion si fuera inmutable) para agruparlos.

Código antes Compra.java

```
public Compra(Usuario usuario, String nombreEspectaculo, LocalDateTime fechaHora,
List<Asiento> asientos) {
   this.usuario = usuario;
   this.nombreEspectaculo = nombreEspectaculo;
   this.fechaHora = fechaHora;
   // ...
}
```

Después

```
// Nuevo record o clase simple
public record DatosFuncion(String nombre, LocalDateTime fecha) {}

// En Compra.java
public Compra(Usuario usuario, DatosFuncion datosFuncion, List<Asiento> asientos) {
   this.usuario = usuario;
   this.nombreEspectaculo = datosFuncion.nombre();
   this.fechaHora = datosFuncion.fecha();
   this.asientos = new ArrayList<>(Objects.requireNonNull(asientos));
}
```

Refactorización: Data Class (EventoFuncion)

- **Code Smell Corregido:** Data Class en EventoFuncion.java.
- **Técnica Aplicada:** *Move Method* (Incorporar comportamiento).

- **Justificación:** EventoFuncion era solo una estructura de datos. Se le agregó comportamiento para formatearse a sí mismo como un String de log legible, quitando esa responsabilidad de otras clases que imprimían el evento.

Código Antes (EventoFuncion.java)

```
public class EventoFuncion {

   // Solo atributos y getters

   public String getTipo() { return tipo; }

   public String getMensaje() { return mensaje; }

   // ...

}
```

Código Después

```
public class EventoFuncion {

   // ... atributos ...


   // Nuevo comportamiento

   public String obtenerResumenLog() {

      return String.format("[%s] %s - %s (%s)",

         fechaHora, tipo.toUpperCase(), mensaje, nombreEspectaculo);

   }

}
```

Refactorización: Shotgun Surgery (Mensajes de Error)

- **Code Smell Corregido:** Shotgun Surgery en manejo de excepciones (strings harcodeados).
- **Técnica Aplicada:** *Replace Magic String with Constant*.
- **Justificación:** Mensajes como "El asiento ya esta agotado" o "no esta disponible" estaban dispersos. Si se quería cambiar el idioma o el texto, había que tocar muchas clases. Se centralizan en una clase MensajesError.

Código Antes (Asiento.java)

```
throw new IllegalStateException("El asiento ya esta agotado");
```

Código Después

```
public class MensajesError {
```

```
    public static final String ASIENTO_AGOTADO = "El asiento solicitado ya se encuentra agotado.";
}

// En Asiento.java
throw new IllegalStateException(MensajesError.ASIENTO_AGOTADO);
```

Refactorización: Middle Man (Facade)

- **Code Smell Corregido:** Middle Man en EnVivoTicketsFacade.
- **Técnica Aplicada:** *Move Method / Inline Method*.
- **Justificación:** La fachada tenía métodos que solo hacían "pasa-manos" sin agregar valor (seguridad, transacciones, simplificación). Para métodos simples de consulta que no requerían orquestación, se permitió que el cliente llame directamente al dominio o se movió la lógica a un controlador más específico, dejando la fachada solo para operaciones complejas de alto nivel (Caso de uso).

Código Antes

```
// En Facade
public void reprogramar(Funcion funcion, LocalDateTime nuevaFecha) {
    funcion.cambiarFecha(nuevaFecha); // Solo pasa la llamada
}
```

Después

```
// Método eliminado de la Fachada.
// El cliente hace:
miFuncion.cambiarFecha(nuevaFecha);
```

Refactorización: Data Class (Usuario)

- **Code Smell Corregido:** Data Class en Usuario.java.
- **Técnica Aplicada:** *Move Method* (Encapsulamiento de lógica de Rol).

- **Justificación:** La clase Usuario exponía el Rol y otros clientes verificaban usuario.getRol() == Rol.ADMIN. Esto viola el encapsulamiento ("Tell, don't ask"). Se agregan métodos de consulta de permisos en el propio usuario.

  Código Antes

  ```
  if (usuario.getRol().equals(Rol.ADMINISTRADOR)) {
     // permitir acción
  }
  ```

  Código Despues

  ```
  public boolean esAdministrador() {
     return this.rol.equals(Rol.ADMINISTRADOR);
  }

  // Uso en cliente
  if (usuario.esAdministrador()) { }
  ```