**DISEÑO DE SOFTWARE**

**PARALELO 3**

**TAREA 4- SISTEMA ENVIVOTICKETS**

**INTEGRANTES:**

**BARRIOS URETA ROBERTO CARLOS**

**MACIAS MENDOZA CHRISTIAN JAVIER**

**TAPIA LOOR PAULO MARCELO**

**PAO II**

**2024-2025**

INDICE

[**Sección A: Identificación de Code Smells** 3](#_Toc189170903)

[1. Primitive Obsession: 3](#_Toc189170904)

[2. Inappropriate Intimacy: 3](#_Toc189170905)

[3. Long Method: 3](#_Toc189170906)

[4. Data Class : 4](#_Toc189170907)

[5. Feature Envy 5](#_Toc189170908)

[6. Null Pointer Risk 6](#_Toc189170909)

[7. Spelling Error 6](#_Toc189170910)

[8. Data Clump 8](#_Toc189170911)

[9. Message Chains 8](#_Toc189170912)

[**10. Refused Bequest** 9](#_Toc189170913)

[**Sección B: Pruebas unitarias** 10](#_Toc189170914)

[**Sección C: Refactorización del código fuente** 10](#_Toc189170915)

# **Sección A: Identificación de Code Smells**

## 1. Primitive Obsession:

**Descripcion:** Se está utilizando un tipo de dato primitivo (int) para representar una entidad más compleja (por ejemplo, una hora). Esto puede llevar a una falta de encapsulación y a una mayor dificultad para agregar comportamientos específicos relacionados con este dato.

**Impacto:** Reduce la expresividad del código. Dificulta futuras modificaciones o mejoras en la lógica de la hora y puede generar código repetitivo con múltiples validaciones en distintos lugares del sistema.

**Tecnica de Refactorizacion adecuada:** Encapsulación en una Clase de Valor (Value Object): En lugar de usar int hora, se podría crear una clase Hora que maneje la validación y representación de la hora.

**Captura:**



## 2. Inappropriate Intimacy:

**Descripcion:** Una clase está accediendo directamente a la implementación interna de otra clase, creando una dependencia innecesaria. En este caso, mapaAsientos parece estar siendo manipulado directamente en lugar de usar métodos encapsulados para realizar operaciones sobre los asientos.

**Impacto:** Aumenta el acoplamiento entre clases, dificultando la mantenibilidad. Si cambia la estructura de mapaAsientos, puede romper múltiples partes del código. Y puede exponer detalles internos de la clase, afectando la encapsulación.

**Tecnica de Refactorizacion adecuada: E**ncapsular el acceso a mapaAsientos en métodos específicos: En lugar de acceder directamente al mapa, se pueden agregar métodos dentro de la clase que lo contiene.

**Captura:**



## 3. Long Method:

**Descripcion:** El método reservarAsiento es demasiado largo y contiene responsabilidades mezcladas. Además, presenta duplicación de cadenas literales ("disponible", "reservado") y validaciones repetitivas.

**Impacto:** Difícil de entender y mantener. Si se requiere modificar la lógica de reserva, se debe actualizar en múltiples lugares. Puede generar inconsistencias en la validación y reserva de asientos.

**Tecnica de Refactorizacion adecuada:** Extraer Métodos (Extract Method): Separar las responsabilidades en métodos más pequeños y reutilizables y eliminar Cadenas Literales Mágicas: Usar constantes o una enumeración para los estados de los asientos.

**Captura:**

A screen shot of a computer code

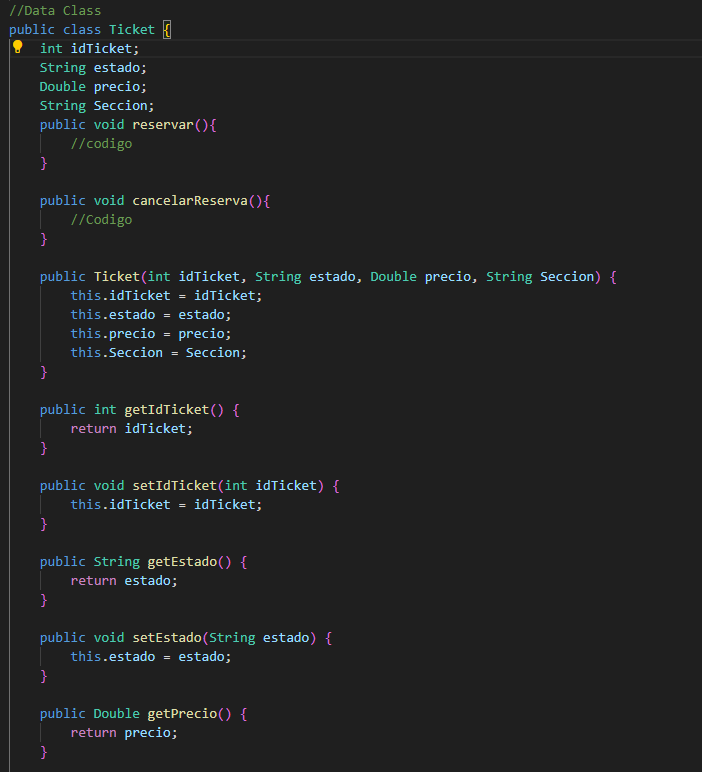
Description automatically generated

## 4. Data Class :

**Descripcion:** Una Data Class es una clase que solo almacena datos sin ninguna lógica de negocio real. Generalmente, contiene solo atributos, constructores y métodos de acceso (getters y setters). En este caso, la clase Ticket solo almacena información sin realizar ninguna operación significativa sobre los datos.

**Impacto:** Este problema genera una falta de encapsulación, lo que obliga a otras clases a manipular los datos directamente, dispersando la lógica de negocio y aumentando el acoplamiento entre clases. Como resultado, cualquier cambio en la lógica de Ticket afectará múltiples partes del código, dificultando el mantenimiento y aumentando el riesgo de errores. Además, la ausencia de métodos específicos para la gestión del estado de los tickets hace que la clase sea poco expresiva y requiera constantes modificaciones externas para su funcionamiento adecuado.

**Tecnica de Refactorizacion adecuada:** Agregar Comportamiento a la Clase, la clase Ticket debería manejar la reserva y cancelación directamente en lugar de depender de otra clase para hacerlo. Usar Métodos Encapsulados en Lugar de Setters Excesivos: En lugar de exponer setEstado(), se deben proporcionar métodos con lógica para cambiar el estado, como reservar() y cancelarReserva().

**Captura:**

## 5. Feature Envy

**Descripcion:** Este code smell ocurre cuando un método de una clase accede a los datos de otra clase con más frecuencia que a los de su propia clase. En este caso, el método recibirNotificacion() usa directamente el atributo nombre, lo que sugiere que probablemente esta funcionalidad debería estar dentro de la clase a la que pertenece nombre.

**Impacto:** Este problema provoca un alto acoplamiento entre clases, lo que hace que la lógica de negocio se disperse en diferentes partes del código. Esto dificulta el mantenimiento, ya que cualquier cambio en la estructura de la clase propietaria de nombre requerirá modificaciones en múltiples lugares. Además, impide que la encapsulación sea efectiva, ya que los datos de una clase son manipulados desde otra.

**Tecnica de Refactorizacion adecuada:** La mejor solución es mover el método a la clase propietaria de nombre, de modo que sea esa clase la que gestione su propia lógica de notificaciones. Esto mejora la cohesión del código y reduce el acoplamiento innecesario.

**Captura:**

A black background with colorful text

Description automatically generated

## 6. Null Pointer Risk

**Descripcion:** Este code smell ocurre cuando existe la posibilidad de que una variable utilizada en un método sea null, lo que puede generar una excepción en tiempo de ejecución. En este caso, la lista incidentes podría no estar inicializada antes de intentar agregar un nuevo incidente.

**Impacto:** Si la lista incidentes es null, se lanzará una NullPointerException, lo que puede interrumpir la ejecución del programa. Además, al no validar la entrada, se podría intentar agregar un incidente null, lo que puede generar inconsistencias en los datos. Este problema reduce la confiabilidad del código y aumenta la probabilidad de fallos en tiempo de ejecución.

**Tecnica de Refactorizacion adecuada:** Se debe asegurar que la lista incidentes esté correctamente inicializada antes de usarla. También se debe validar que el parámetro incidente no sea null antes de agregarlo.

**Captura:**

A black background with colorful text

Description automatically generated

## 7. Spelling Error

**Descripcion:** Este code smell ocurre cuando hay errores tipográficos en los nombres de clases, variables o métodos. En este caso, el nombre de la clase FecadeSystem parece ser una incorrecta escritura de FacadeSystem, lo que puede generar confusión entre los desarrolladores y dificultar la lectura del código. También hay un error en la variable x dentro del System.out.println(), lo que probablemente cause una compilación incorrecta.

**Impacto:**

**Tecnica de Refactorizacion adecuada:** Los errores de escritura en los identificadores pueden hacer que el código sea menos legible y provocar confusión entre los desarrolladores. También pueden generar errores de compilación si una variable mal escrita es referenciada sin ser definida. Estos problemas afectan la mantenibilidad y la confiabilidad del código.

**Captura:**

A computer screen shot of a program code

Description automatically generated

## 8. Data Clump

**Descripcion:** Este code smell ocurre cuando un grupo de variables relacionadas aparecen juntas en múltiples lugares del código, lo que indica que deberían encapsularse en una clase propia. En este caso, la lista usuarios es un conjunto de datos que probablemente esté siendo utilizada en otras partes del sistema de manera similar.

**Impacto:** Tener datos dispersos en diferentes lugares dificulta la cohesión y la reutilización del código. Si se necesita agregar funcionalidad específica para los usuarios, como filtrado o validación, se debe modificar múltiples partes del código en lugar de hacerlo en una única clase. Esto aumenta el riesgo de inconsistencias y reduce la mantenibilidad.

**Tecnica de Refactorizacion adecuada:** Se debe encapsular la lista usuarios dentro de una clase especializada, como GestorUsuarios, que maneje la gestión de usuarios de manera centralizada.

**Captura:**

****

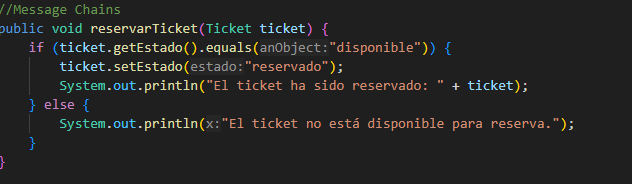
## 9. Message Chains

**Descripcion:** Este code smell ocurre cuando un objeto accede a los métodos de otro objeto de forma encadenada, lo que indica una dependencia excesiva entre las clases. En este caso, ticket.getEstado().equals("disponible") es un ejemplo de Message Chain, ya que reservarTicket() depende de la estructura interna de Ticket.

**Impacto:** Este problema hace que el código sea frágil y difícil de mantener, ya que si la implementación de Ticket cambia, todas las partes del código que dependen de getEstado() se verán afectadas. También rompe el principio de encapsulamiento, exponiendo la lógica interna de Ticket en otra clase, lo que dificulta futuras modificaciones.

**Tecnica de Refactorizacion adecuada:** Para reducir la dependencia entre clases, se debe delegar la verificación del estado dentro de la clase Ticket. En lugar de llamar directamente a getEstado(), se debe agregar un método estaDisponible() que encapsule la lógica de verificación.

**Captura:**

****

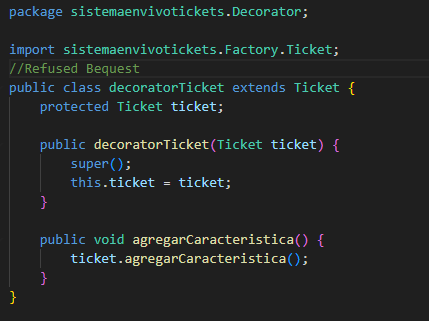
## **10. Refused Bequest**

**Descripcion:** Este code smell ocurre cuando una subclase hereda métodos o atributos de una clase base, pero no los utiliza de manera significativa, lo que indica que la herencia no es la estrategia adecuada. En este caso, decoratorTicket extiende Ticket, pero parece que no aprovecha los métodos y atributos de Ticket, sino que en su lugar contiene una referencia directa a una instancia de Ticket.

**Impacto:** El uso incorrecto de la herencia genera confusión y hace que el código sea más difícil de entender y mantener. Si la subclase no necesita todos los métodos y atributos de la superclase, esto puede llevar a un diseño ineficiente y violar el principio de sustitución de Liskov, ya que decoratorTicket no se comporta completamente como un Ticket.

**Tecnica de Refactorizacion adecuada:** La solución es reemplazar la herencia con composición. En lugar de hacer que decoratorTicket extienda Ticket, se debe hacer que implemente una interfaz o contenga una referencia a Ticket sin necesidad de herencia.

**Captura:**



# **Sección B: Pruebas unitarias**

# **Sección C: Refactorización del código fuente**

**LINK: <https://github.com/RoberB1/Tarea2--EnVivoTickets>**