**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](#_Toc189171387)

[1. Primitive Obsession: 3](#_Toc189171388)

[2. Inappropriate Intimacy: 3](#_Toc189171389)

[3. Long Method: 3](#_Toc189171390)

[4. Data Class : 4](#_Toc189171391)

[5. Feature Envy 5](#_Toc189171392)

[6. Null Pointer Risk 6](#_Toc189171393)

[7. Spelling Error 6](#_Toc189171394)

[8. Data Clump 8](#_Toc189171395)

[9. Message Chains 8](#_Toc189171396)

[10. Refused Bequest 9](#_Toc189171397)

[**Sección B: Pruebas unitarias** 10](#_Toc189171398)

[**Sección C: Refactorización del código fuente** 12](#_Toc189171399)

# **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 usar la clase LocalTime.



## 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

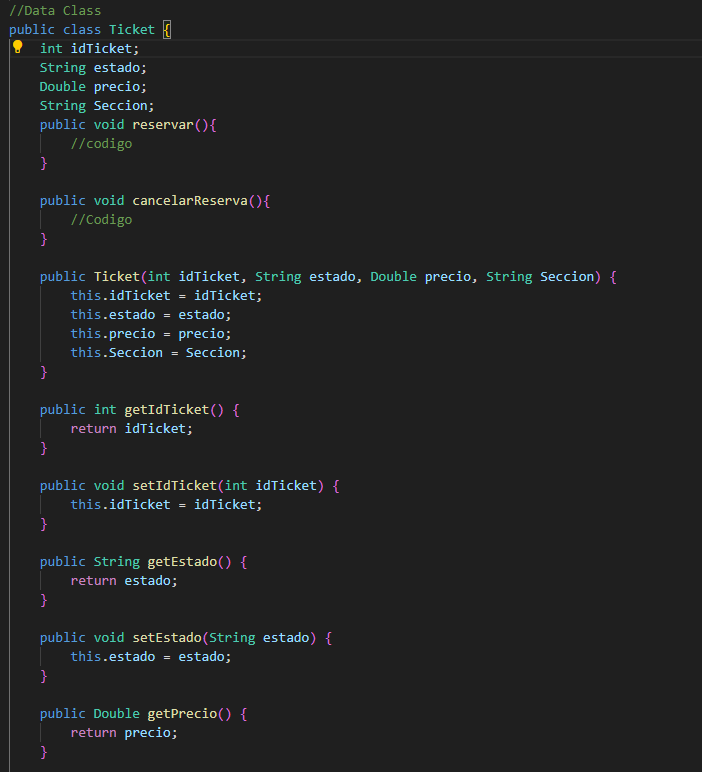
Description automatically generated

## 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:**

****

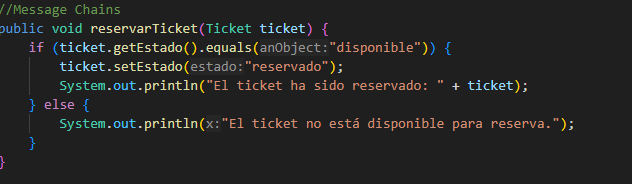
## 9. Message Chains

**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:**

****

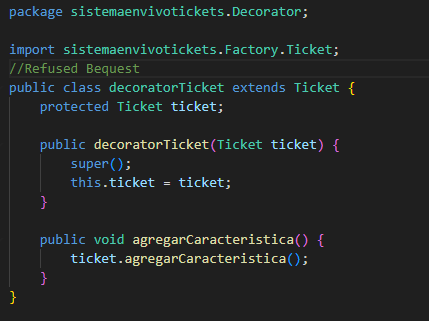
## 10. Refused Bequest

**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**

Tabla

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ID** | **Clase** | **Método** | **Propósito** | **Precondiciones** | **Entradas** | **Salidas Esperadas** |
| 001 | Evento | agregarObservador | Verificar que un observador se agrega correctamente | Lista de observadores inicializada | Observador válido | Observador agregado |
| 002 | Evento | agregarObservador | Manejo de error al agregar observador nulo | Lista de observadores inicializada | null | Excepción lanzada |
| 003 | Evento | eliminarObservador | Verificar que un observador se elimina correctamente | Lista con al menos un observador | Observador existente | Observador eliminado |
| 004 | Evento | eliminarObservador | Intento de eliminar observador inexistente | Lista sin el observador | Observador no existente | Lista sin cambios |
| 005 | FacadeSystem | verDisponibilidad | Verificar disponibilidad de asientos de un evento | Evento y función válidos | Evento y función | Impresión de disponibilidad |
| 006 | FacadeSystem | reservarTicket | Reservar un ticket disponible | Ticket en estado "disponible" | Ticket válido | Estado del ticket "reservado" |
| 007 | FacadeSystem | reservarTicket | Intentar reservar un ticket ya reservado | Ticket en estado "reservado" | Ticket válido | Mensaje de error |
| 008 | FacadeSystem | realizarPago | Verificar que se realiza un pago correctamente | Sistema funcional | Monto positivo | Mensaje de pago exitoso |
| 009 | Funcion | mostrarDisponibilidad | Verificar la impresión de disponibilidad de asientos | Mapa de asientos inicializado | - | Lista de asientos en consola |
| 010 | Funcion | reservarAsiento | Reservar asiento disponible | Asiento en estado "disponible" | Número de asiento | Asiento pasa a estado "reservado" |
| 011 | Funcion | reservarAsiento | Intentar reservar asiento ya ocupado | Asiento en estado "reservado" | Número de asiento | Mensaje de error |
| 012 | Soporte | registrarIncidente | Verificar registro de un incidente | Lista de incidentes inicializada | Descripción de incidente | Incidente agregado a la lista |
| 013 | Soporte | escalarIncidente | Escalar un incidente correctamente | Lista de incidentes inicializada | Descripción de incidente | Mensaje de escalamiento |
| 014 | Usuario | recibirNotificacion | Verificar recepción de notificación | Usuario con nombre y email definidos | Mensaje de notificación | Mensaje impreso con nombre |
| 015 | Ticket | reservar | Verificar reserva de un ticket | Ticket en estado "disponible" | Ticket válido | Ticket en estado "reservado" |
| 016 | Ticket | cancelarReserva | Cancelar reserva de un ticket reservado | Ticket en estado "reservado" | Ticket válido | Ticket en estado "disponible" |
| 017 | TicketConAlimento | agregarCaracteristica | Verificar adición de característica de alimento | Ticket válido | Ticket decorado | Mensaje "Alimento agregado al ticket" |
| 018 | TicketConBebida | agregarCaracteristica | Verificar adición de característica de bebida | Ticket válido | Ticket decorado | Mensaje "Bebida agregada al ticket" |
| 019 | TicketConEstacionamiento | agregarCaracteristica | Verificar adición de característica de estacionamiento | Ticket válido | Ticket decorado | Mensaje "Estacionamiento agregado al ticket" |
| 020 | Notificación | agregarObservador | Verificar que un observador se agrega correctamente | Observador válido | Observador agregado | Observador agregado al evento |
| 021 | Notificación | eliminarObservador | Verificar eliminación de observador | Observador en la lista | Observador existente | Observador eliminado del evento |
| 022 | Notificación | cambiarProgramacion | Verificar cambio de programación | Evento válido | Detalle de cambio | Mensaje de notificación de cambio |
| 023 | Subject | recibirNotificación | Verificar recepción de notificación y propagación | Observadores agregados | Mensaje de notificación | Observadores reciben el mensaje |
| 024 | Subject | notificarCambios | Verificar notificación a observadores | Observadores agregados | Mensaje de notificación | Mensaje propagado a observadores |

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

## 1. Encapsulación en una Clase de Valor (Value Object)

**Problema Inical:** Se usaba un tipo primitivo (int) para representar una entidad compleja como una hora, lo que hacía que la lógica de validación y manipulación estuviera dispersa en diferentes partes del código.

**Beneficios obtenidos tras la refactorización:**

* Mayor encapsulación y centralización de la lógica de validación.
* Mayor claridad y facilidad para modificar la representación de la hora.
* Código más expresivo y menos propenso a errores.

**Capturas:**

## 2. Encapsular el Acceso a Datos (Encapsulate Collection)

**Problema Inicial:** Se accedía directamente a mapaAsientos desde otras clases, lo que generaba una fuerte dependencia entre ellas y exponía detalles internos de la implementación.

**Beneficios obtenidos tras la refactorización:**

* Reducción del acoplamiento entre clases.
* Mejor mantenimiento y seguridad de la información.
* Código más estructurado y reutilizable.

**Capturas:**

## 3. Extraer Métodos (Extract Method)

**Problema Inicial:** El método reservarAsiento() era demasiado largo y mezclaba varias responsabilidades, lo que dificultaba su mantenimiento.

**Beneficios obtenidos tras la refactorización:**

* Separación de responsabilidades dentro del método.
* Código más legible y modular.
* Facilidad para reutilizar funciones y hacer pruebas unitarias.

**Capturas:**

## 4. Agregar Comportamiento a la Clase (Introduce Behavior in Class)

**Problema Inicial:** La clase Ticket solo contenía datos (Data Class), sin lógica asociada para manejar su propio estado.

**Beneficios obtenidos tras la refactorización:**

* Mejor cohesión, ya que Ticket ahora maneja su propia lógica.
* Reducción del acoplamiento con otras clases.
* Código más fácil de extender en el futuro.

**Capturas:**

## 5. Mover Método a la Clase Correcta (Move Method)

**Problema Inicial:** El método recibirNotificacion() usaba nombre, lo que indicaba que pertenecía a la clase Usuario, no a su ubicación original.

**Beneficios obtenidos tras la refactorización:**

* Mayor cohesión, ya que el comportamiento se encuentra donde realmente pertenece.
* Reducción de dependencias innecesarias.
* Mejor organización del código.

**Capturas:**

**A screenshot of a computer program

Description automatically generated**

## 6. Manejo Seguro de Nulos (Null Object Pattern)

**Problema Inicial:** El código no verificaba si incidentes estaba inicializado, lo que podía provocar NullPointerException.

**Beneficios obtenidos tras la refactorización:**

* Mayor estabilidad del código.
* Eliminación de errores en tiempo de ejecución.
* Mejor control sobre el estado de los datos.

**Capturas:**

**A screenshot of a computer

Description automatically generated**

## 7. Reemplazar Herencia con Composición (Replace Inheritance with Composition)

**Problema Inicial:** La clase decoratorTicket heredaba de Ticket sin usar la mayoría de sus métodos, lo que indicaba un mal uso de la herencia.

**Beneficios obtenidos tras la refactorización:**

* Eliminación de acoplamiento innecesario.
* Código más flexible y reutilizable.
* Mejor mantenimiento y extensibilidad.

**Capturas:**

## 8. Corregir Errores Tipográficos (Fix Spelling Errors)

**Problema Inicial:** Errores en el nombre de la clase FecadeSystem y en la variable x.

**Beneficios obtenidos tras la refactorización:**

* Código más legible y profesional.
* Eliminación de posibles errores de compilación.
* Mejora en la comprensión del código.

**Capturas:**

## 9. Reemplazo de Cadenas de Mensajes (Hide Delegate - Message Chains)

**Problema Inicial:** El método reservarTicket() accedía a ticket.getEstado().equals("disponible"), lo que generaba una cadena de mensajes. Esto creaba una fuerte dependencia con la estructura interna de Ticket, lo que hacía que cualquier cambio en Ticket afectara este método.

**Beneficios obtenidos tras la refactorización:** El código es más flexible y menos propenso a cambios en la estructura interna de Ticket. Se encapsula la lógica dentro de la clase correspondiente, mejorando la cohesión y reduciendo el acoplamiento. Ahora, en lugar de preguntar el estado del ticket desde fuera, el propio Ticket expone un método estaDisponible() que maneja internamente la lógica de disponibilidad.

**Capturas:**

## 10. Reemplazo de Herencia por Composición (Replace Inheritance with Delegation - Refused Bequest)

**Problema Inicial:** La clase decoratorTicket heredaba de Ticket pero no utilizaba todos sus métodos y atributos, lo que indicaba un mal uso de la herencia. Esto generaba Refused Bequest, ya que la subclase ignoraba gran parte de la funcionalidad heredada.

**Beneficios obtenidos tras la refactorización:** Se eliminó la herencia innecesaria, haciendo que decoratorTicket use composición en lugar de herencia. Esto reduce el acoplamiento con Ticket, evita que decoratorTicket herede métodos innecesarios y permite una mejor extensibilidad y reutilización del código sin depender de la jerarquía de clases.

**Capturas:**

**LINK:** [**https://github.com/RoberB1/Tarea2--EnVivoTickets**](https://github.com/RoberB1/Tarea2--EnVivoTickets)