**DISEÑO DE SOFTWARE**

**TAREA #4**

**Informe sobre Refactoring y Pruebas**

**Integrantes del Grupo:**

- Ariel Arias

- Erick Murillo

- Cesar Mera

- Adrián Toledo

# **Introducción**

El equipo llevará a cabo la mejora del código del proyecto asignado mediante la aplicación de técnicas de refactorización. Para ello, analizará el código fuente con el fin de identificar al menos 10 code smells, justificando su impacto en la calidad del software y corrigiéndolos con técnicas adecuadas.Asimismo, desarrollará pruebas unitarias para al menos 12 clases del sistema, considerando tanto casos normales como escenarios de falla. Se emplearán herramientas de control de versiones para gestionar los cambios y documentar cada mejora en el repositorio de GitHub.A través de esta actividad, el equipo fortalecerá sus habilidades en desarrollo colaborativo y en la aplicación de fundamentos de ciencias computacionales para la producción de software de calidad.

## **Objetivos Específicos**

1. Reorganizar el código fuente de un sistema mediante técnicas de refactorización para simplificar los mantenimientos futuros.
2. Emplear herramientas informáticas en el control de versiones y la ejecución de pruebas unitarias para garantizar la calidad del software en un entorno colaborativo.

# **Resultado de Aprendizaje**

1. Funcionar efectivamente como miembro o líder de un equipo involucrado en actividades apropiadas para la disciplina del programa.
2. Aplicar teoría de ciencias computacionales y fundamentos de desarrollo de software para generar soluciones computacionales eficientes.

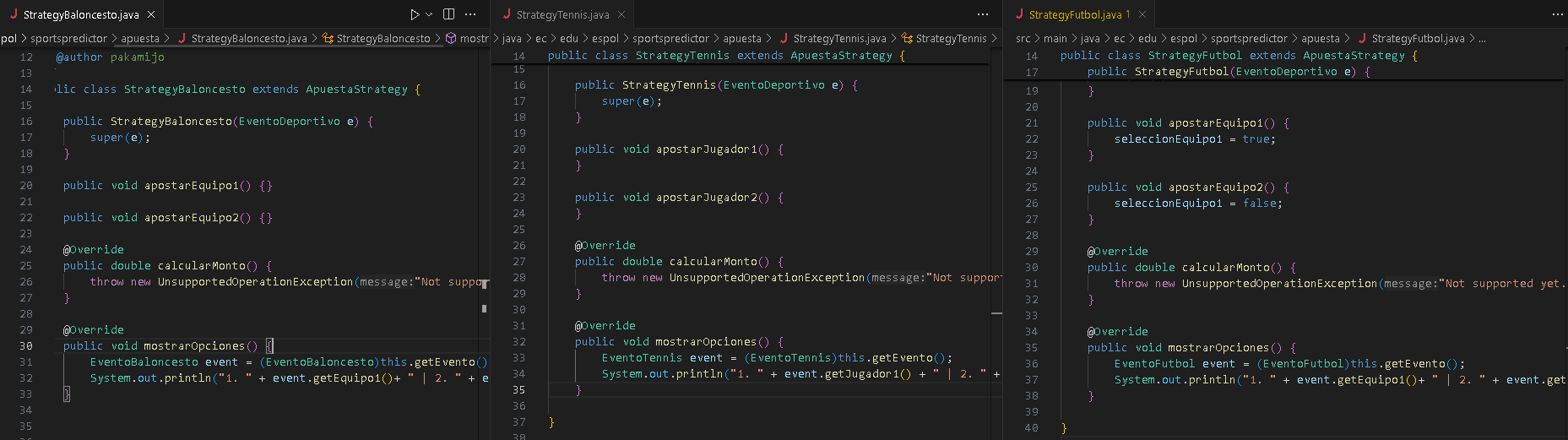
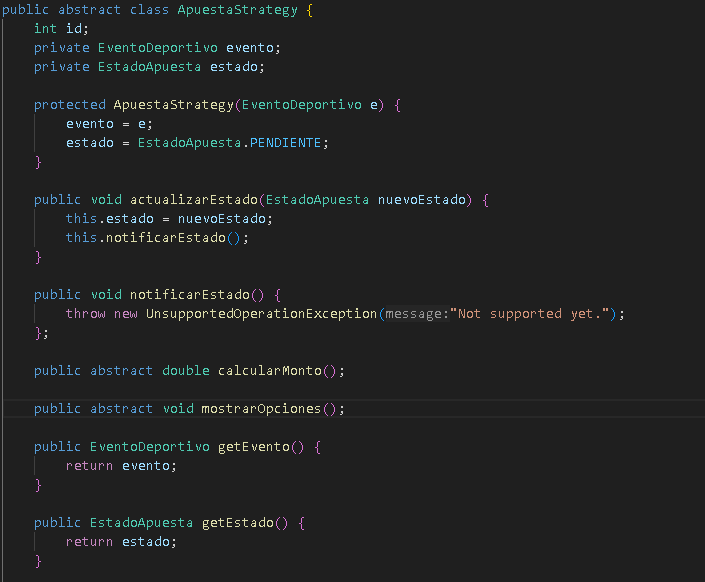
# Sección A: Identificación de Code Smells

1. Código Duplicado (Duplicated Code)

En las clases StrategyFutbol, StrategyBaloncesto y StrategyTennis, los métodos mostrarOpciones() tienen una estructura muy similar, solo cambiando el tipo de evento. Esto genera duplicación de código.

Impacto: Dificulta el mantenimiento, ya que cualquier cambio en la lógica de mostrar opciones debe replicarse en todas las clases. Además, aumenta el riesgo de inconsistencias.

Técnica de refactorización: Extraer Superclase o Extraer Método para centralizar la lógica común en la clase base ApuestaStrategy.

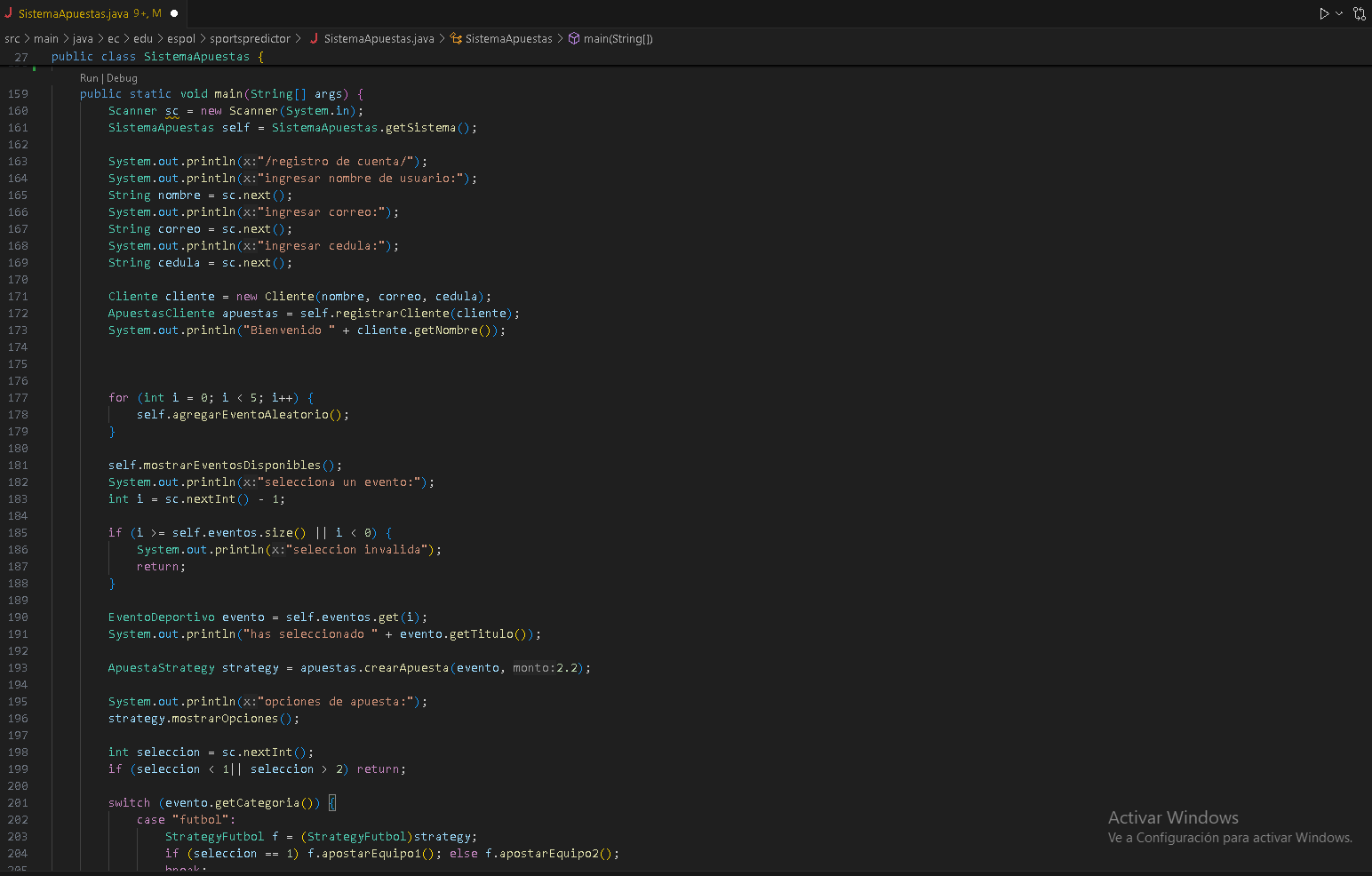


2. Métodos Largos (Long Method)

El método main en la clase SistemaApuestas es demasiado largo y realiza múltiples tareas (registro de cliente, creación de eventos, selección de apuestas, etc.).

Reduce la legibilidad y dificulta la comprensión del flujo del programa. También complica las pruebas unitarias.

Técnica de refactorización: Extraer Método para dividir el método main en métodos más pequeños y específicos.

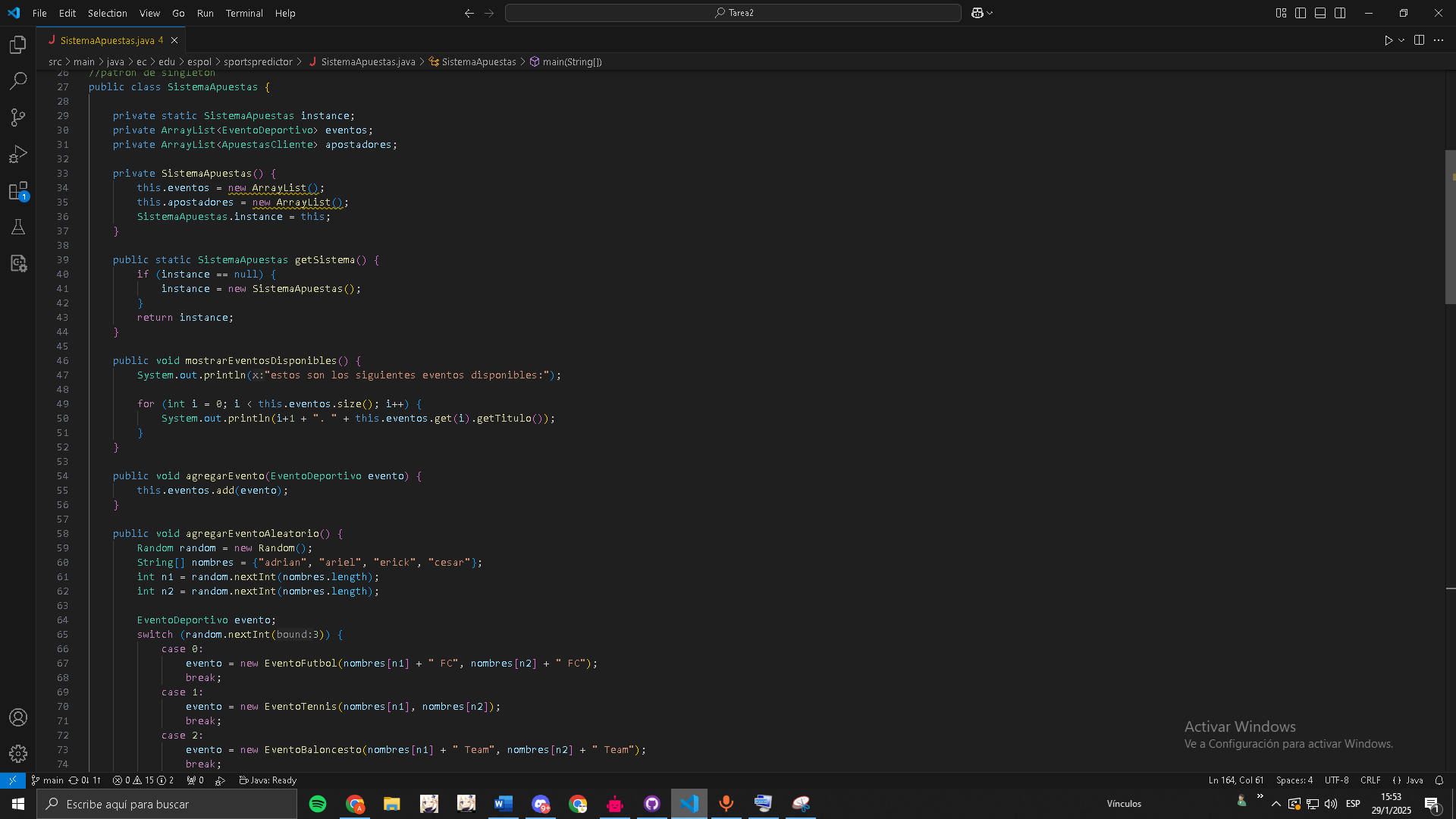


3. Clases Grandes (Large Class)

La clase SistemaApuestas tiene demasiadas responsabilidades (gestión de eventos, registro de clientes, interacción con el usuario, etc.).

Dificulta el mantenimiento y la escalabilidad, ya que cualquier cambio puede afectar múltiples funcionalidades.

Técnica de refactorización: Extract Class para separar las responsabilidades en clases más pequeñas y cohesivas.



4. Switch Statements

En el método crearApuesta de la clase ApuestasCliente, se utiliza un switch para determinar el tipo de estrategia de apuesta. Esto viola el principio abierto/cerrado.

Dificulta la extensión del código, ya que agregar un nuevo tipo de apuesta requiere modificar este método.

Técnica de refactorización: Reemplazar Condicional con Polimorfismo para delegar la creación de estrategias a un factory.



5. Métodos No Implementados (Unimplemented Methods)

Varios métodos, como calcularMonto() en las estrategias de apuestas y actualizar() en las notificaciones, no están implementados y lanzan excepciones.

Genera código incompleto y puede causar errores en tiempo de ejecución.

Técnica de refactorización: Implementar Métodos o Eliminar Métodos No Utilizados si no son necesarios.

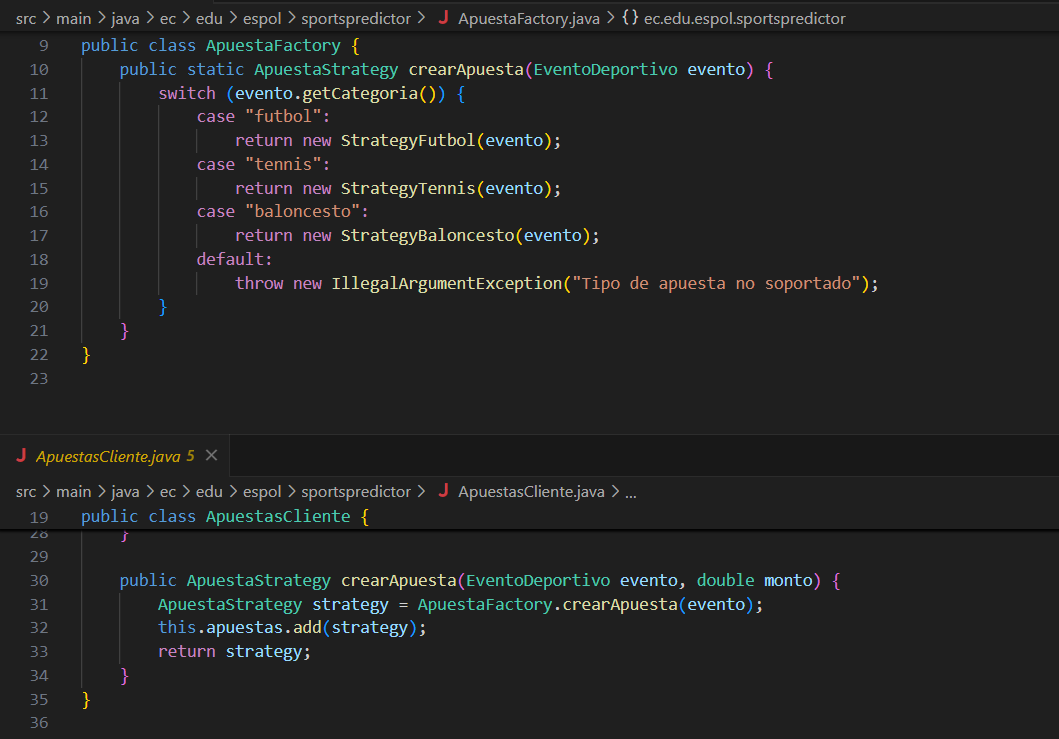


6. Acoplamiento Excesivo (Feature Envy)

La clase ApuestasCliente accede directamente a los detalles de las estrategias de apuestas y eventos deportivos, lo que indica un alto acoplamiento.

Reduce la flexibilidad y dificulta la reutilización del código.

Técnica de refactorización: Mover Método para transferir la lógica relacionada con las apuestas a las clases de estrategia.

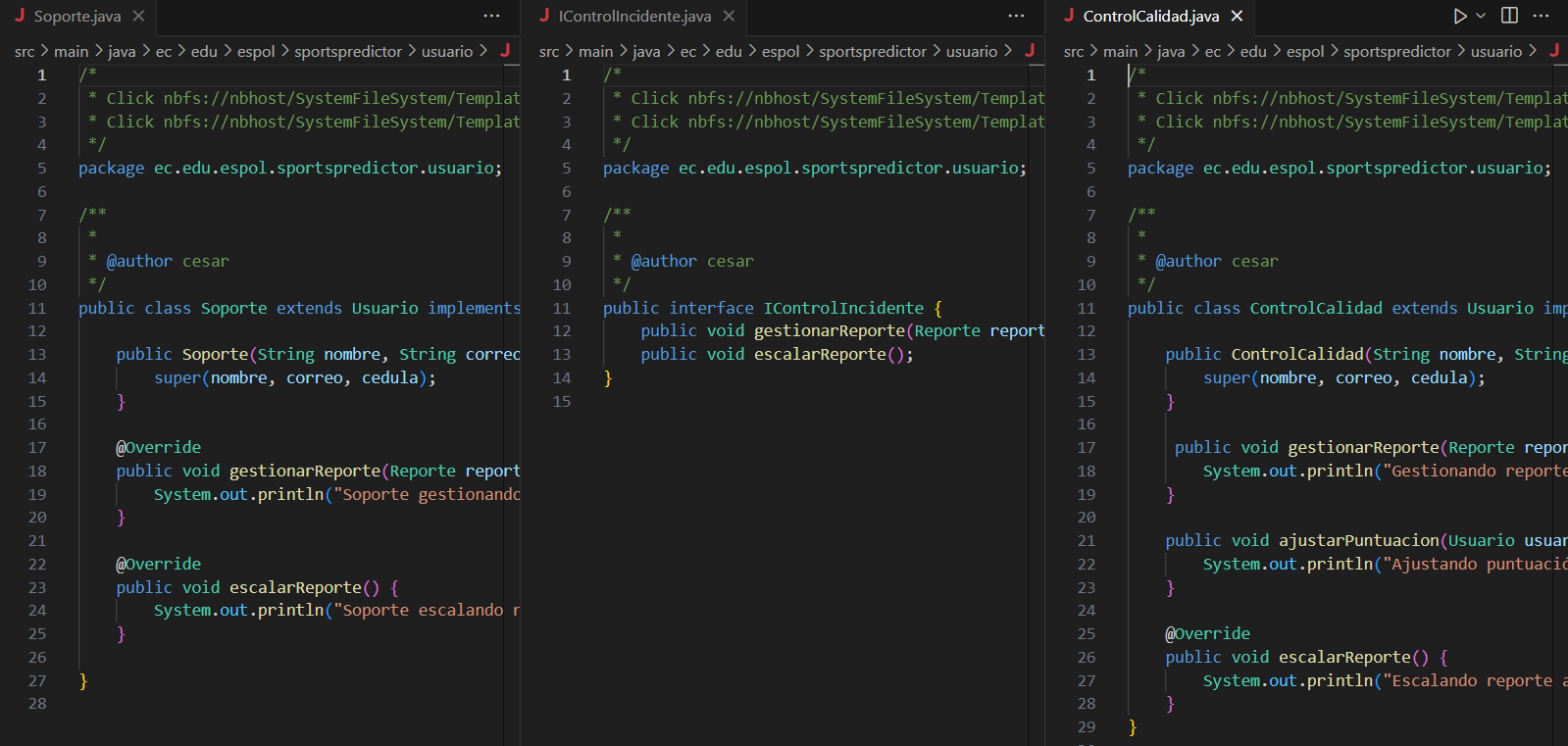


7. Uso Incorrecto de Herencia

Descripción del problema: La clase ControlCalidad y Soporte implementan la interfaz IControlIncidente, pero no todas las implementaciones son necesarias o coherentes.

Impacto: Puede llevar a una jerarquía de clases confusa y poco clara.

Técnica de refactorización: Reemplazar Herencia con Delegación para simplificar la estructura.

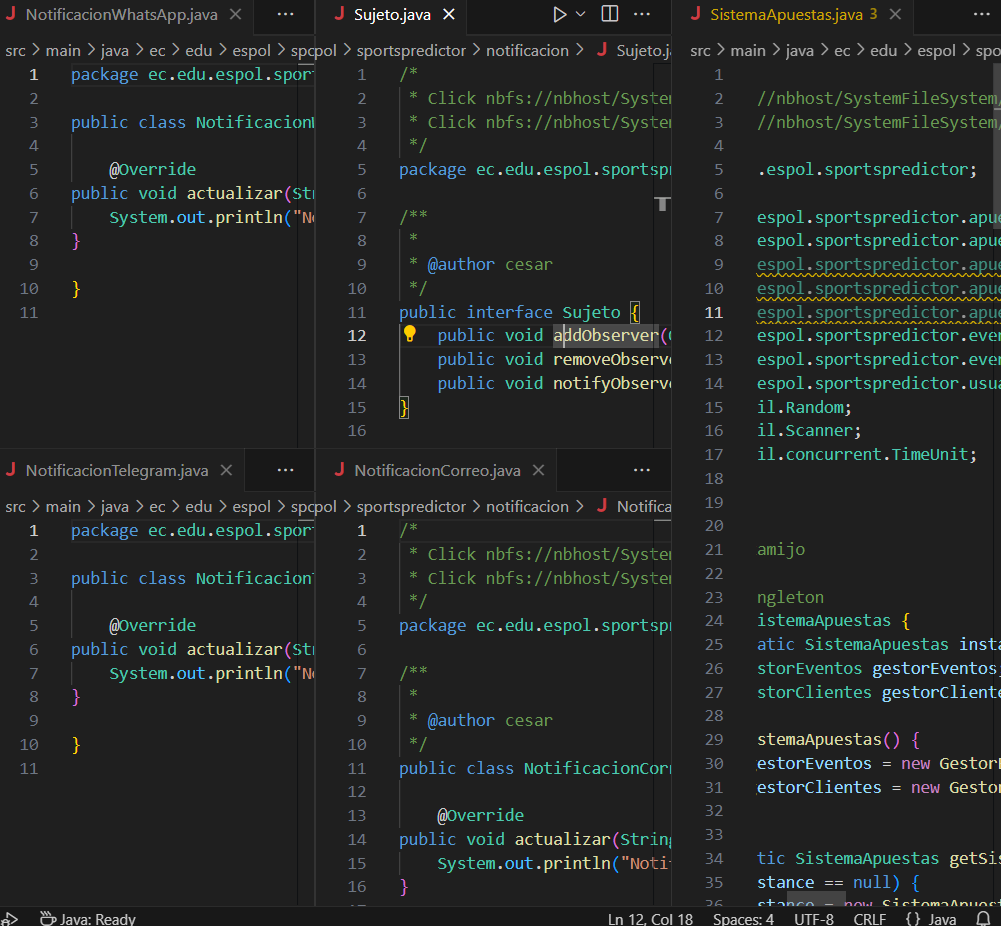


8. Código Muerto (Dead Code)

Descripción del problema: La clase NotificacionCorreo, NotificacionTelegram y NotificacionWhatsApp no están siendo utilizadas en el flujo principal del programa.

Impacto: Aumenta la complejidad del código sin aportar valor.

Técnica de refactorización: Eliminar Código Muerto para simplificar el código.

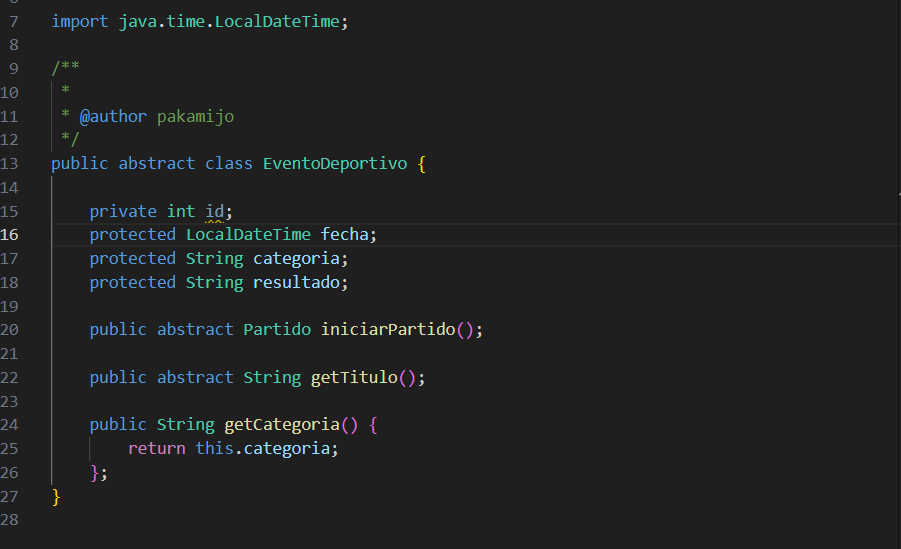


9. Falta de Encapsulamiento

Descripción del problema: Los atributos de la clase EventoDeportivo (como categoria y resultado) son accesibles directamente desde otras clases.

Impacto: Dificulta el control sobre el estado interno de los objetos y puede llevar a inconsistencias.

Técnica de refactorización: Encapsular Campo para restringir el acceso directo a los atributos.

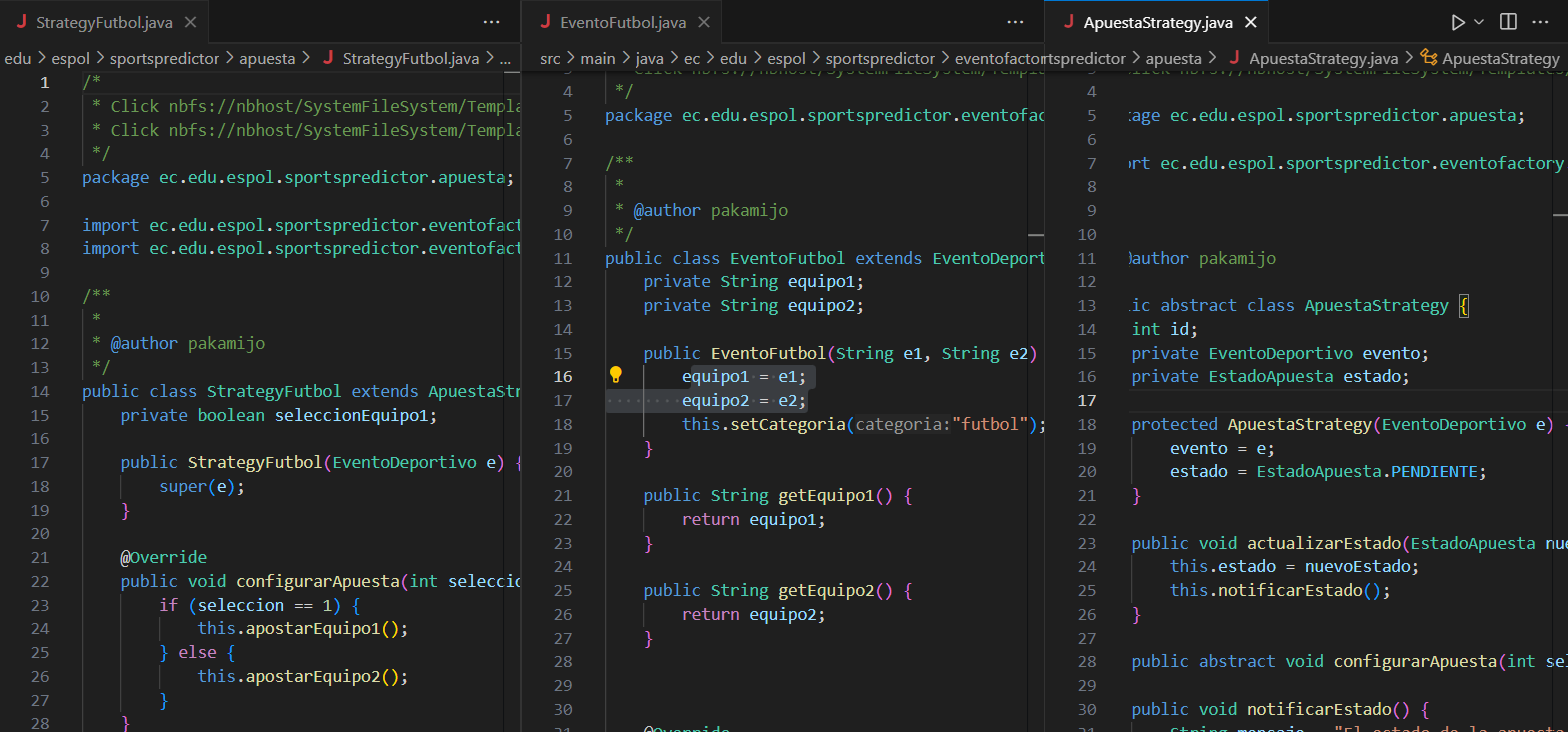


10. Nombres No Descriptivos

Descripción del problema: Nombres como e1, e2, j1, j2 en los constructores de eventos no son descriptivos.

Impacto: Reduce la legibilidad y comprensión del código.

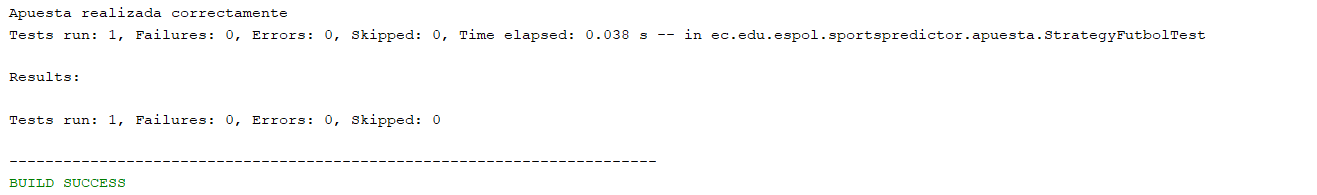
Técnica de refactorización: Renombrar Variables para usar nombres más significativos.



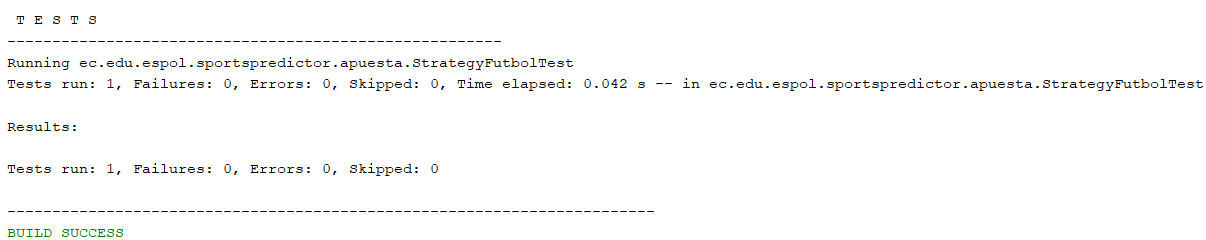
# **Sección B: Pruebas unitarias**

Strategy Futbol Test

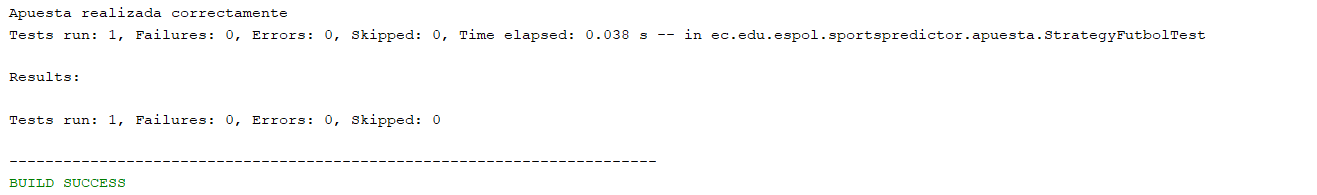
|  |  |
| --- | --- |
| ID: | 001 |
| Propósito: | Probar que el método configurarApuesta funciona correctamente con entradas válidas. |
| Precondiciones: | Instancia válida de StrategyFutbol con un evento de fútbol. |
| Entradas y salidas esperadas: | seleccion=1 |
| Resultado: | apuesta realizada correctamente. |



|  |  |
| --- | --- |
| ID: | 002 |
| Propósito: | Validar que el método configurarApuesta maneja selecciones inválidas. |
| Precondiciones: | Nd |
| Entradas y salidas esperadas: | seleccion=3 |
| Resultado: | Excepción IllegalArgumentException |

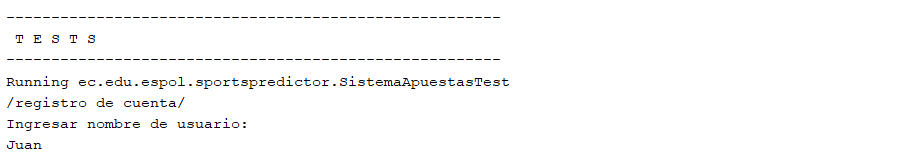


|  |  |
| --- | --- |
| ID: | 003 |
| Propósito: | Probar que el método configurarApuesta funciona correctamente con entradas válidas. |
| Precondiciones: | Instancia válida de StrategyFutbol con un evento de fútbol. |
| Entradas y salidas esperadas: | seleccion=2 |
| Resultado: | apuesta realizada correctamente. |

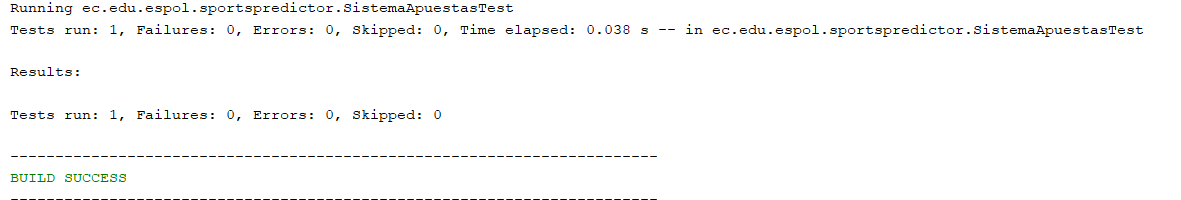


Sistema Apuestas Test

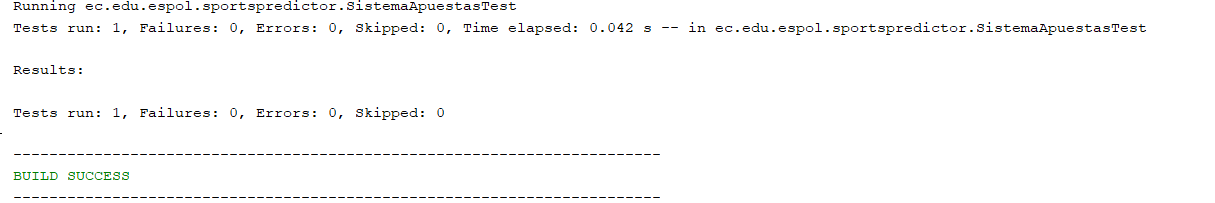
|  |  |
| --- | --- |
| ID: | 004 |
| Propósito: | Verificar que el sistema registra correctamente a un cliente. |
| Precondiciones: | SistemaApuestas debe estar correctamente inicializado. |
| Entradas y salidas esperadas: | Nombre: "Juan" |
| Resultado: | Cliente registrado con el nombre "Juan". |



|  |  |
| --- | --- |
| ID: | 005 |
| Propósito: | Validar que el sistema de apuestas devuelve un gestor de eventos no nulo. |
| Precondiciones: | SistemaApuestas debe estar correctamente inicializado. |
| Entradas y salidas esperadas: | Llamada a getGestorEventos(). |
| Resultado: | Un objeto GestorEventos no nulo. |

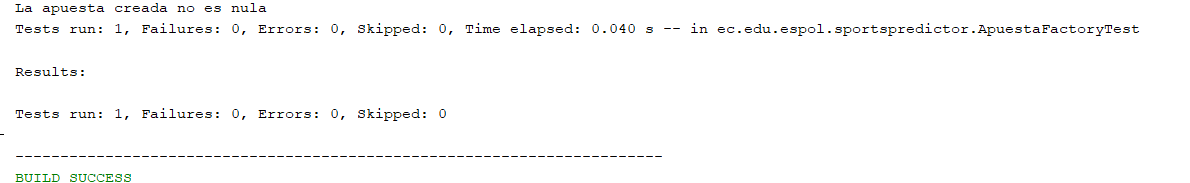


|  |  |
| --- | --- |
| ID: | 006 |
| Propósito: | Asegurar que el sistema de apuestas devuelve un gestor de clientes válido. |
| Precondiciones: | SistemaApuestas debe estar correctamente inicializado. |
| Entradas y salidas esperadas: | Llamada a getGestorClientes(). |
| Resultado: | Un objeto GestorClientes no nulo. |

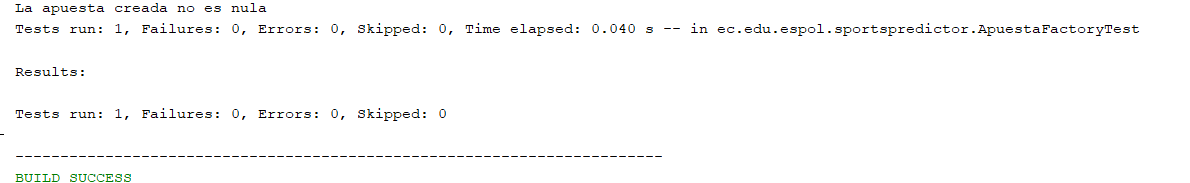


Apuesta Factory Test

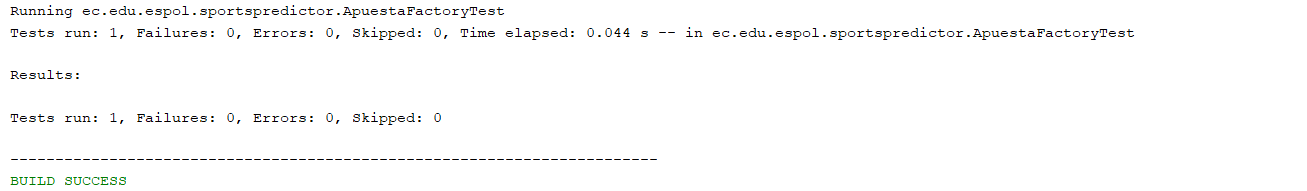
|  |  |
| --- | --- |
| ID: | 007 |
| Propósito: | Verificar que la fábrica crea correctamente una instancia de StrategyFutbol cuando la categoría es "futbol" |
| Precondiciones: | Existe un objeto EventoDeportivo configurado con categoría "futbol" |
| Entradas y salidas esperadas: | Entrada: EventoDeportivo con categoría "futbol" |
| Resultado: | La apuesta creada no es nula |



|  |  |
| --- | --- |
| ID: | 008 |
| Propósito: | Verificar que la fábrica crea correctamente una instancia de StrategyFutbol cuando la categoría es "tennis" |
| Precondiciones: | Existe un objeto EventoDeportivo configurado con categoría "tennis" |
| Entradas y salidas esperadas: | Entrada: EventoDeportivo con categoría "tennis" |
| Resultado: | La apuesta creada no es nula |

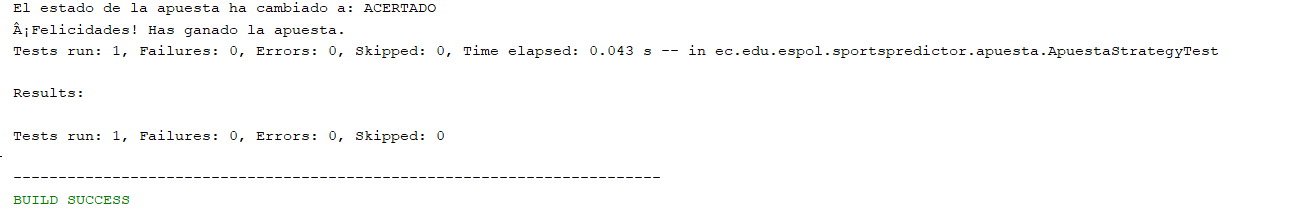


|  |  |
| --- | --- |
| ID: | 009 |
| Propósito: | Verificar que la fábrica lanza una excepción cuando se proporciona una categoría no soportada |
| Precondiciones: | Existe un objeto EventoDeportivo configurado con una categoría inválida ("ajedrez") |
| Entradas y salidas esperadas: | EventoDeportivo con categoría "ajedrez" |
| Resultado: | Se lanza IllegalArgumentException |

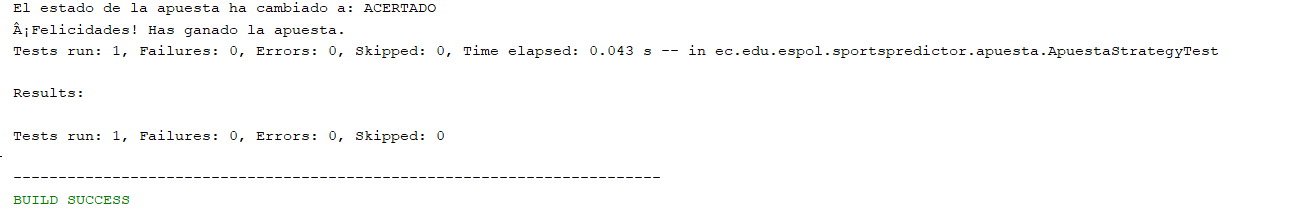


Apuesta Strategy Test

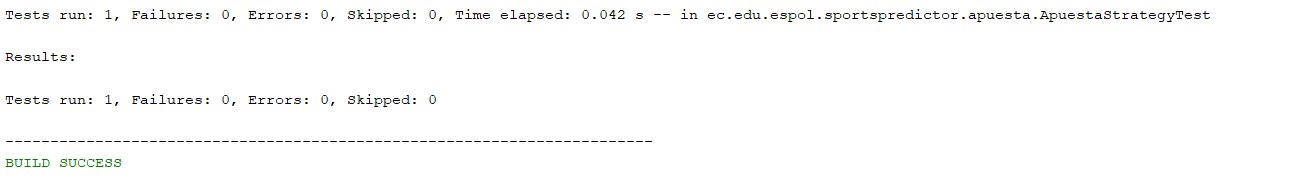
|  |  |
| --- | --- |
| ID: | 011 |
| Propósito: | Verificar que el método actualizarEstado cambia correctamente el estado de la apuesta |
| Precondiciones: | Existe un objeto EventoDeportivo |
| Entradas y salidas esperadas: | EstadoApuesta.ACERTADO |
| Resultado: | El estado de la apuesta se actualiza correctamente a ACERTADO |



|  |  |
| --- | --- |
| ID: | 010 |
| Propósito: | Verificar que el método notificarEstado muestra los mensajes correctos cuando la apuesta es acertada |
| Precondiciones: | Existe un objeto EventoDeportivo |
| Entradas y salidas esperadas: | EstadoApuesta.ACERTADO |
| Resultado: | Se muestra el mensaje de cambio de estado  Se muestra el mensaje de felicitación por ganar la apuesta |

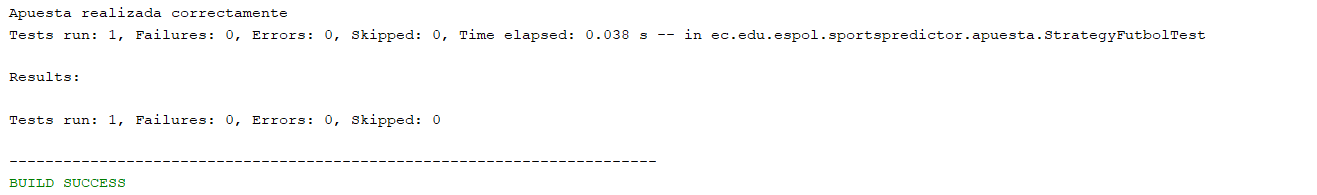


|  |  |
| --- | --- |
| ID: | 012 |
| Propósito: | Verificar que el método getEvento retorna correctamente el evento deportivo asociado a la apuesta |
| Precondiciones: | Existe un objeto EventoDeportivo con categoría "futbol" |
| Entradas y salidas esperadas: | Llamada al método getEvento() |
| Resultado: | El evento recuperado no es null |

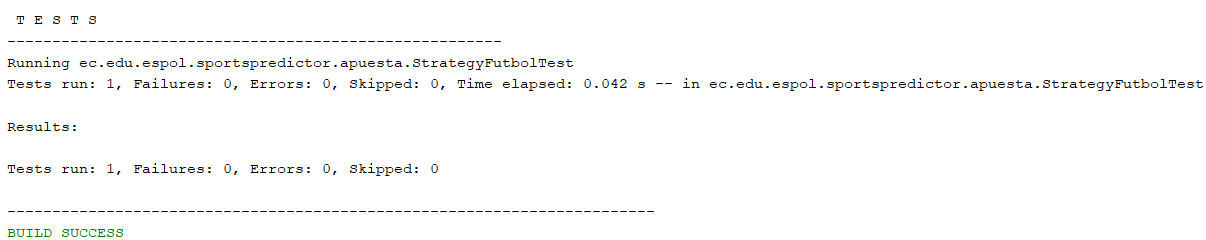


Strategy Tennis Test

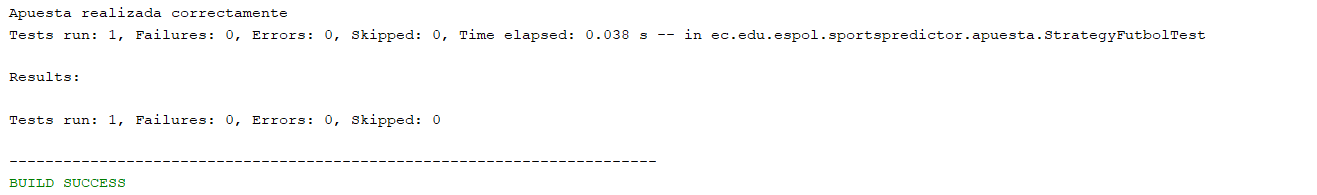
|  |  |
| --- | --- |
| ID: | 013 |
| Propósito: | Verificar que el método configurarApuesta maneja correctamente las opciones válidas (1 y 2) |
| Precondiciones: | Instancia válida de StrategyTennisl con un evento de tennis |
| Entradas y salidas esperadas: | Entrada 1: opcionSeleccionada = 1  Salida 1: apostarEquipo1() retorna true  Entrada 2: opcionSeleccionada = 2  Salida 2: apostarEquipo1() retorna false | | Resultado: | - La apuesta se configura correctamente para ambos equipos |
| Resultado: | apuesta realizada correctamente. |



|  |  |
| --- | --- |
| ID: | 014 |
| Propósito: | Verificar que el método configurarApuesta rechaza valores fuera del rango válido |
| Precondiciones: | Nd |
| Entradas y salidas esperadas: | seleccion=3 |
| Resultado: | Excepción IllegalArgumentException |



|  |  |
| --- | --- |
| ID: | 015 |
| Propósito: | Verificar que el método configurarApuesta maneja correctamente la opción 0 |
| Precondiciones: | Instancia válida de StrategyTennis con un evento de tennis |
| Entradas y salidas esperadas: | Entrada: opcionSeleccionada = 0 |
| Resultado: | IllegalArgumentException |



|  |  |
| --- | --- |
| ID: | 015 |
| Propósito: | Verificar que el método configurarApuesta maneja correctamente la opción 0 |
| Precondiciones: | Instancia válida de StrategyTennis con un evento de tennis |
| Entradas y salidas esperadas: | Entrada: opcionSeleccionada = 0 |
| Resultado: | IllegalArgumentException |

GestorClientes Test

|  |  |
| --- | --- |
| ID: | 016 |
| Propósito: | Verificar que el método registrarCliente agrega correctamente un cliente a la lista de apostadores. |
| Precondiciones: | Instancia válida de GestorClientes y un objeto Cliente. |
| Entradas y salidas esperadas: | Entrada: Cliente cliente = new Cliente("Juan")  Salida: void |
| Resultado: | Realizado correctamente |

A black screen with white text

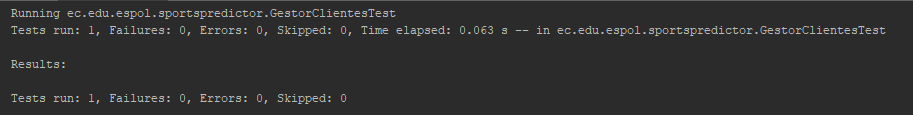
Description automatically generated

|  |  |
| --- | --- |
| ID: | 017 |
| Propósito: | Verificar que el método registrarCliente lanza una excepción cuando el cliente es null. |
| Precondiciones: | Instancia válida de GestorClientes |
| Entradas y salidas esperadas: | Entrada: Cliente cliente = null  Salida: NullPointerException |
| Resultado: | void |

A black background with white text

Description automatically generated

|  |  |
| --- | --- |
| ID: | 018 |
| Propósito: | Verificar que el método getApostadores devuelve una lista vacía cuando no hay clientes registrados. |
| Precondiciones: | Instancia válida de GestorClientes sin clientes registrados. |
| Entradas y salidas esperadas: | Entrada: Ninguna  Resultado: Lista vacia |
| Resultado: | Realizado correctamente |

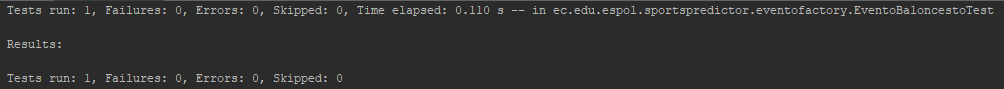


EventoBaloncesto Test

|  |  |
| --- | --- |
| ID: | 019 |
| Propósito: | Verificar que el método getTitulo devuelve el título correcto del evento de baloncesto. |
| Precondiciones: | Instancia válida de EventoBaloncesto con equipos "Lakers" y "Warriors". |
| Entradas y salidas esperadas: | Entrada: equipo1 = "Lakers", equipo2 = "Warriors"  Resultado: "Baloncesto: Lakers vs Warriors" |
| Resultado: | Realizado correctamente |



|  |  |
| --- | --- |
| ID: | 020 |
| Propósito: | Verificar que el método iniciarPartido devuelve una instancia de PartidoBaloncesto. |
| Precondiciones: | Instancia válida de EventoBaloncesto |
| Entradas y salidas esperadas: | Entrada: ninguna  Salida: Instancia de PartidoBaloncesto. |
| Resultado: | Realizado correctamente |

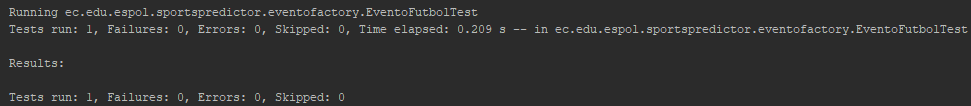


|  |  |
| --- | --- |
| ID: | 021 |
| Propósito: | Verificar que el constructor lanza una excepción cuando uno de los equipos es null. |
| Precondiciones: | Ninguna |
| Entradas y salidas esperadas: | Entrada: ningunaequipo1 = "Lakers", equipo2 = null  Salida: NullPointerException |
| Resultado: | void |

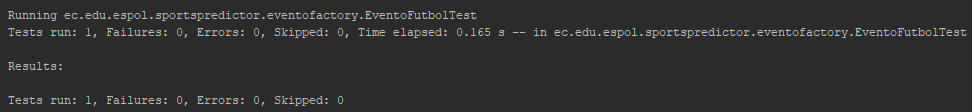


EventosFutbol Test

|  |  |
| --- | --- |
| ID: | 022 |
| Propósito: | Verificar que el método getTitulo devuelve el título correcto del evento de fútbol. |
| Precondiciones: | Instancia válida de EventoFutbol con equipos "Barcelona" y "Real Madrid". |
| Entradas y salidas esperadas: | Entrada: equipo1 = "Barcelona", equipo2 = "Real Madrid"  Salida: "Futbol: Barcelona vs Real Madrid" |
| Resultado: | Realizado exitosamente |



|  |  |
| --- | --- |
| ID: | 023 |
| Propósito: | Verificar que el método iniciarPartido devuelve una instancia de PartidoFutbol. |
| Precondiciones: | Instancia válida de EventoFutbol. |
| Entradas y salidas esperadas: | Entrada: Ninguna  Salida: Instancia de PartidoFutbol |
| Resultado: | Realizado exitosamente |



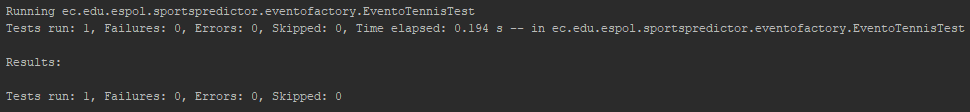
|  |  |
| --- | --- |
| ID: | 024 |
| Propósito: | Verificar que el constructor lanza una excepción cuando uno de los equipos es null. |
| Precondiciones: | Ninguna |
| Entradas y salidas esperadas: | Entrada: equipo1 = "Barcelona", equipo2 = null  Salida: NullPointerException |
| Resultado: | Realizado exitosamente |

A screenshot of a computer

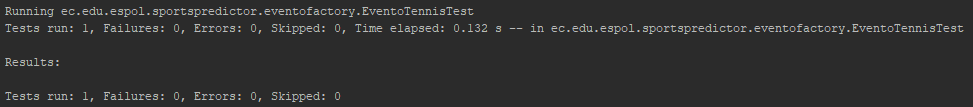
Description automatically generated

EventoTennis Test

|  |  |
| --- | --- |
| ID: | 025 |
| Propósito: | Verificar que el método getTitulo devuelve el título correcto del evento de tenis. |
| Precondiciones: | Instancia válida de EventoTennis con jugadores "Nadal" y "Federer". |
| Entradas y salidas esperadas: | Entrada: jugador1 = "Nadal", jugador2 = "Federer"  Salida: "Tennis Nadal vs Federer" |
| Resultado: | Realizado exitosamente |



|  |  |
| --- | --- |
| ID: | 026 |
| Propósito: | Verificar que el método iniciarPartido devuelve una instancia de PartidoTennis. |
| Precondiciones: | Instancia válida de EventoTennis. |
| Entradas y salidas esperadas: | Entrada: Ninguna  Salida: Instancia de PartidoTennis |
| Resultado: | Realizado exitosamente |



|  |  |
| --- | --- |
| ID: | 027 |
| Propósito: | Verificar que el constructor lanza una excepción cuando uno de los jugadores es null. |
| Precondiciones: | Ninguna |
| Entradas y salidas esperadas: | Entrada: jugador1 = "Nadal", jugador2 = null  Salida: NullPointerException |
| Resultado: | void |

A grey background with white text

Description automatically generated

GestorEventos Test

|  |  |
| --- | --- |
| ID: | 028 |
| Propósito: | Verificar que el método agregarEvento agrega correctamente un evento a la lista. |
| Precondiciones: | Instancia válida de GestorEventos y un objeto EventoDeportivo |
| Entradas y salidas esperadas: | Entrada: EventoDeportivo evento = new EventoFutbol("Barcelona", "Real Madrid")  Salida: La lista eventos debe contener un elemento. |
| Resultado: | void |

A screen shot of a computer

Description automatically generated

|  |  |
| --- | --- |
| ID: | 029 |
| Propósito: | Verificar que el método seleccionarEvento devuelve null cuando el índice es inválido. |
| Precondiciones: | Instancia válida de GestorEventos sin eventos registrados. |
| Entradas y salidas esperadas: | Entrada: indice = 5  Salida: null |
| Resultado: | void |

A screen shot of a computer

Description automatically generated

|  |  |
| --- | --- |
| ID: | 030 |
| Propósito: | Verificar que el método agregarEventoAleatorio agrega un evento a la lista. |
| Precondiciones: | Instancia válida de GestorEventos. |
| Entradas y salidas esperadas: | Entrada: Ninguna  Salida: La lista eventos debe contener un elemento. |
| Resultado: | void |

A screen shot of a computer

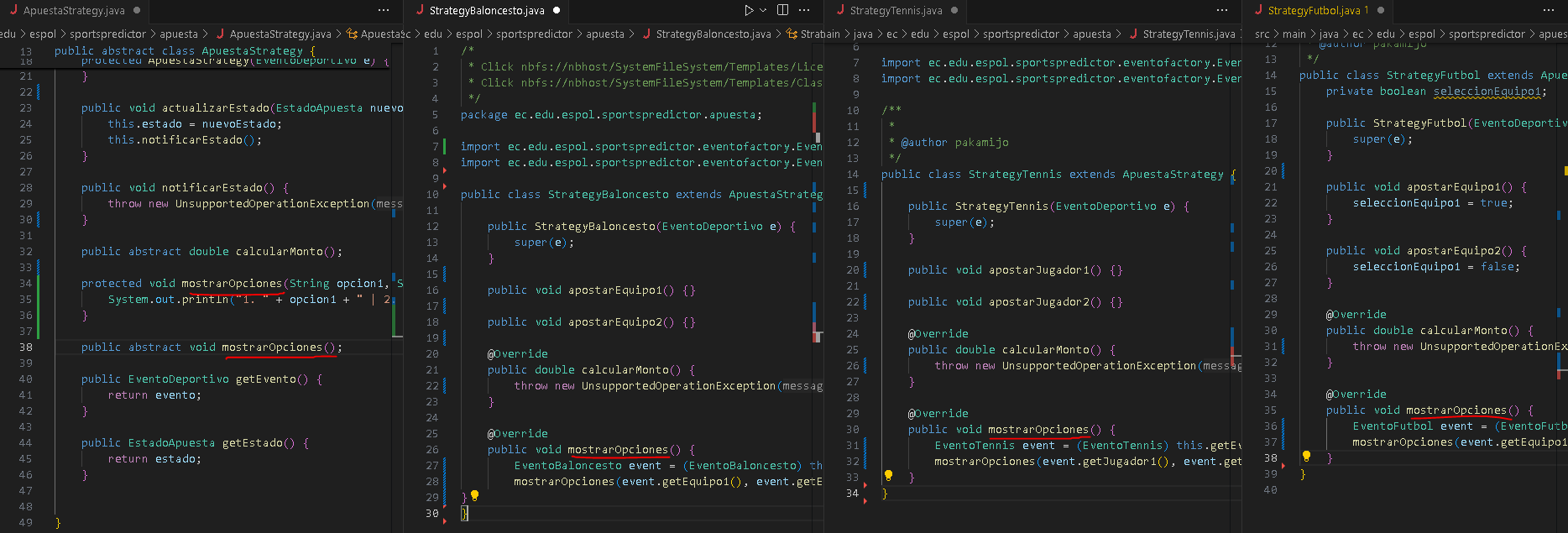
Description automatically generated

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

1. Refactoring para Código Duplicado (Duplicated Code)

Extraer Método: Se creó un método protegido en la clase base para mostrar las opciones: **protected void mostrarOpciones(String opcion1, String**  **opcion2)**   
 Pull Up Method: Se movió el método mostrarOpciones(String opcion1, String opcion2) a la clase base ApuestaStrategy.

Las subclases ahora llaman a este método con los nombres específicos de las opciones:

 Beneficios de la Refactorización

Eliminación de Duplicación: El código repetido en las subclases se eliminó, centralizando la lógica en la clase base.

Mantenibilidad: Si necesitas cambiar la forma en que se muestran las opciones (por ejemplo, agregar un formato diferente), solo debes modificar el método mostrarOpciones en la clase base.

Legibilidad: El código es más limpio y fácil de entender, ya que la lógica común está en un solo lugar.

Flexibilidad: Nuevas subclases pueden reutilizar el método mostrarOpciones sin necesidad de duplicar código.

1. Refactoring para el long method

Extraer Métodos: Se extrajeron las tareas específicas del método main en métodos más pequeños:

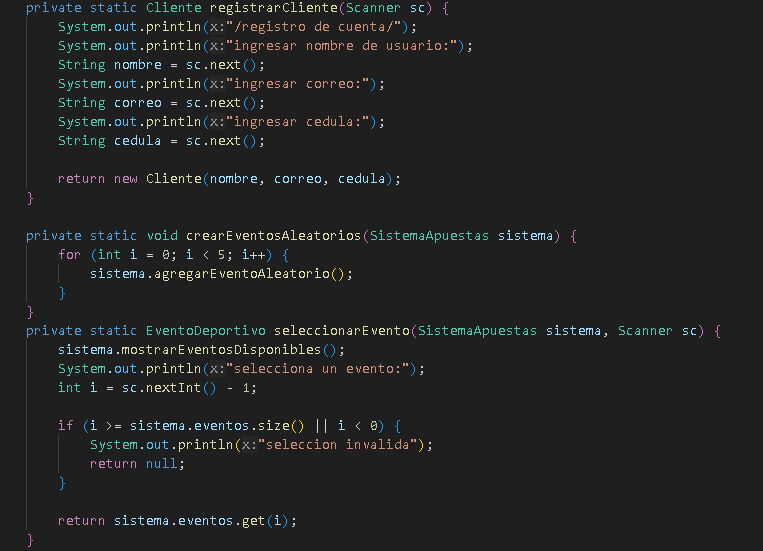
registrarCliente: Maneja el registro del cliente.

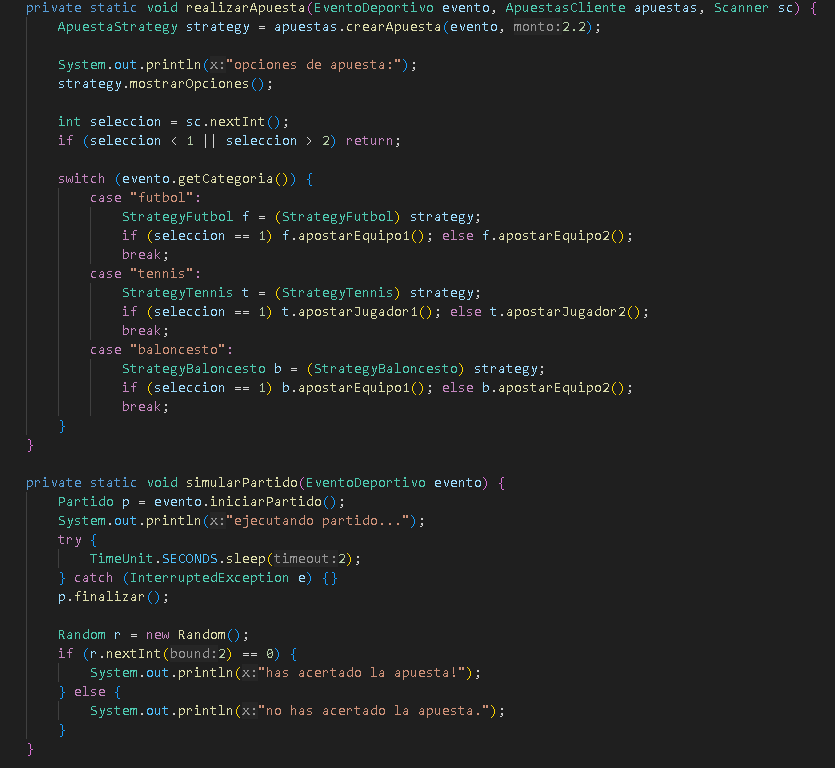
crearEventosAleatorios: Crea eventos deportivos aleatorios.

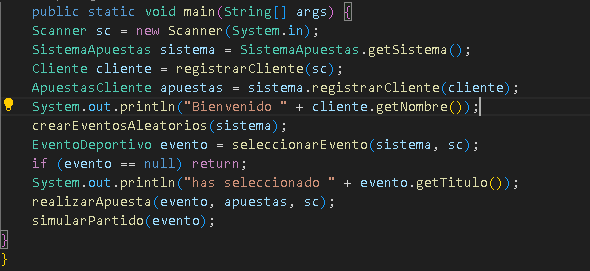
seleccionarEvento: Permite al usuario seleccionar un evento.

realizarApuesta: Gestiona la lógica de la apuesta.

simularPartido: Simula el partido y muestra el resultado.







Beneficios de la Refactorización

Legibilidad: El método main ahora es más corto y fácil de entender.

Mantenibilidad: Cada método tiene una responsabilidad única, lo que facilita su modificación.

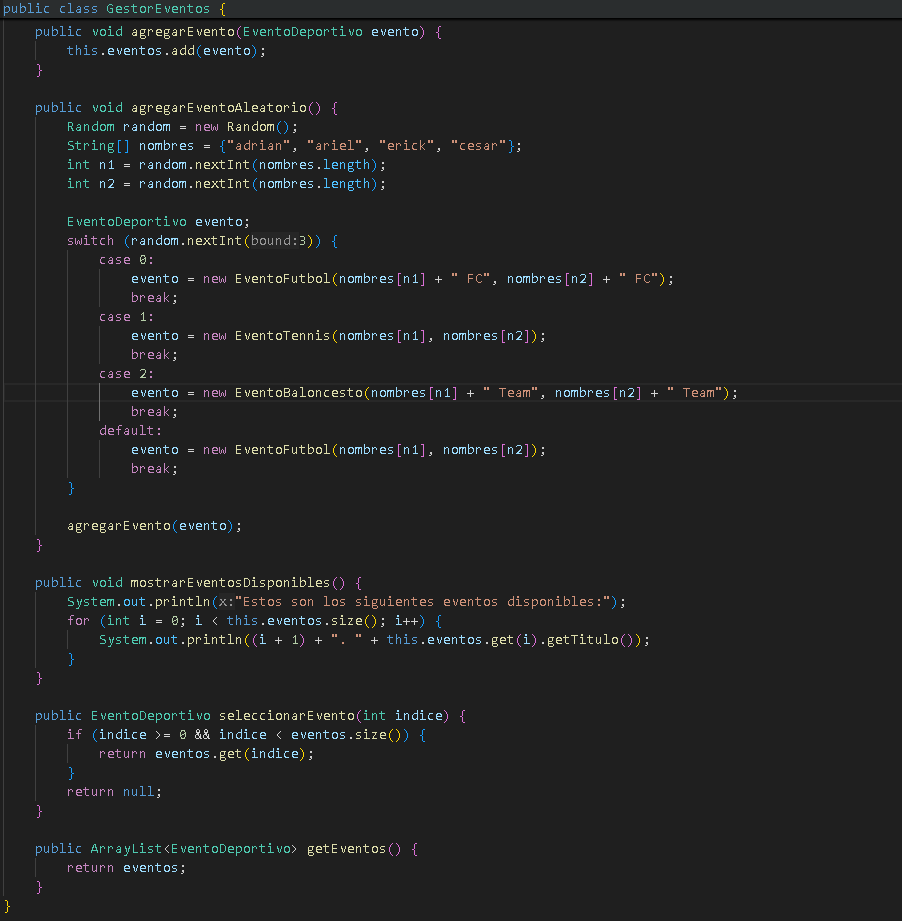
Pruebas Unitarias: Los métodos pequeños y específicos son más fáciles de probar de manera aislada.

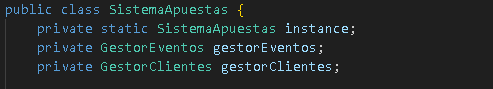
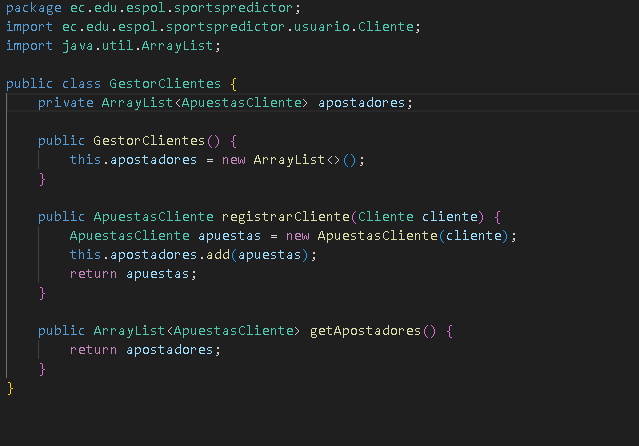
Flexibilidad: Si necesitas agregar nuevas funcionalidades, puedes hacerlo en métodos separados sin afectar el flujo principal.

1. Refactoring de Large Class

Extract Class

La clase SistemaApuestas ahora delega la gestión de eventos y clientes a las clases GestorEventos y GestorClientes, respectivamente.

.



Beneficios de la Refactorización

División de Responsabilidades: La clase SistemaApuestas ahora delega la gestión de eventos y clientes a las clases GestorEventos y GestorClientes, respectivamente.

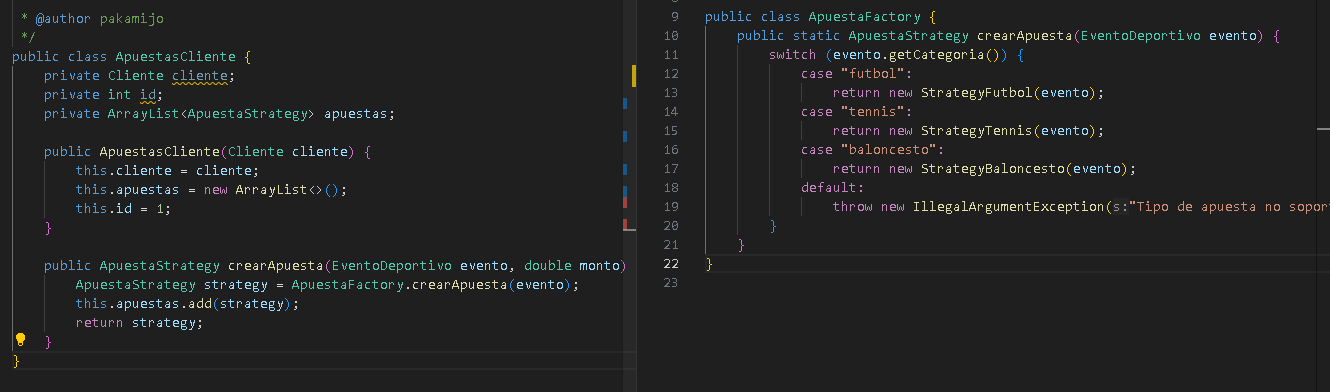
Esto mejora la cohesión y reduce el acoplamiento.

Mantenibilidad: Cada clase tiene una responsabilidad única, lo que facilita su modificación y prueba.

Legibilidad: El código es más limpio y fácil de entender.

1. Refactorizacion para el Switch Statement

Reemplazar Condicional con Polimorfismo para delegar la creación de estrategias a un factory



Beneficios de la Refactorización

Polimorfismo: El uso de un Factory Method para crear estrategias de apuesta la necesidad de un switch en la clase ApuestasCliente.

Esto hace que el código sea más extensible y cumple con el principio abierto/cerrado.

Mantenibilidad: Cada clase tiene una responsabilidad única, lo que facilita su modificación y prueba.

Legibilidad: El código es más limpio y fácil de entender.

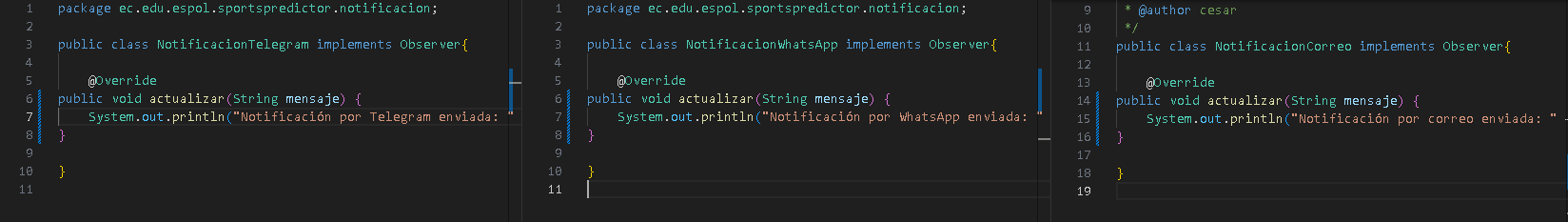
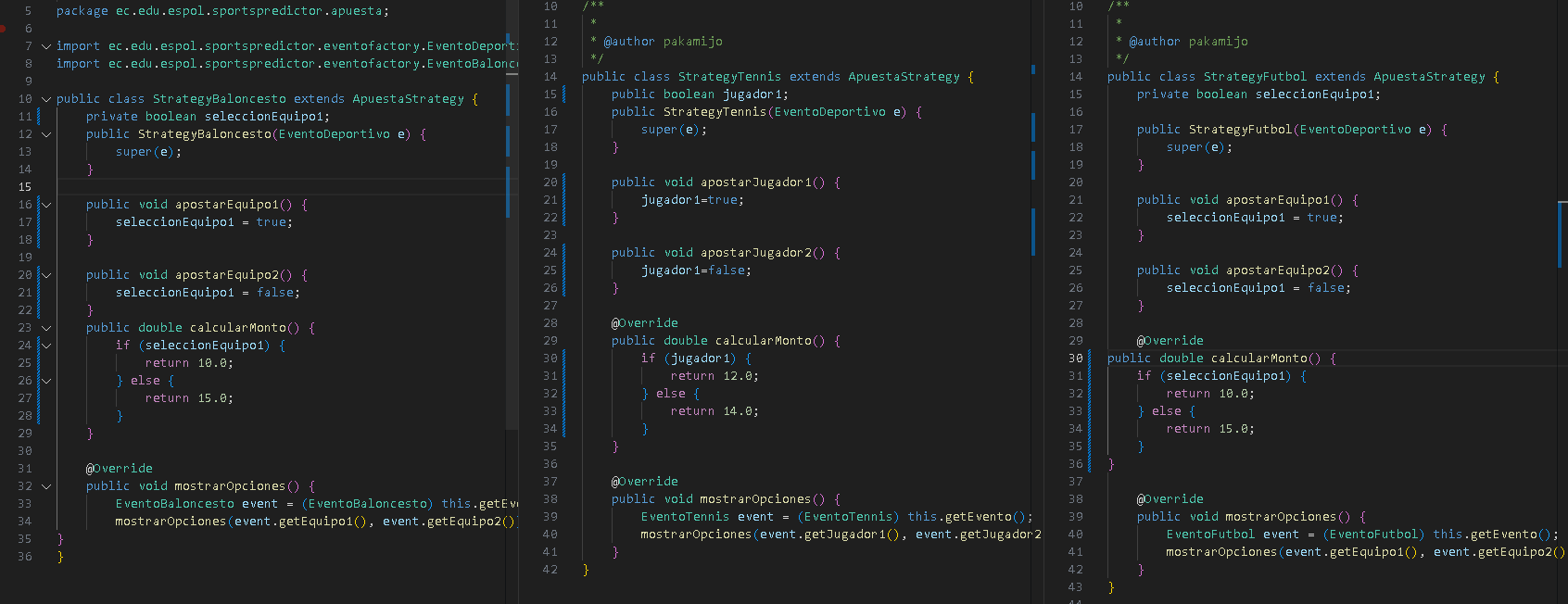
1. Implementar Métodos No Implementados

Los siguientes métodos no están implementados y lanzan excepciones: calcularMonto() en las clases StrategyFutbol, StrategyBaloncesto y StrategyTennis.

otificarEstado() en la clase ApuestaStrategy.

actualizar() en las clases NotificacionCorreo, NotificacionTelegram y NotificacionWhatsApp.

Implementaremos los métodos no implementados con lógica real para que cumplan con su propósito.

 Beneficios de la Refactorización

Devolución de Estrategia: realizarApuesta ahora retorna la estrategia (ApuestaStrategy) para su uso en simularPartido.

Validación: Si la estrategia es null, el programa muestra un mensaje y finaliza.

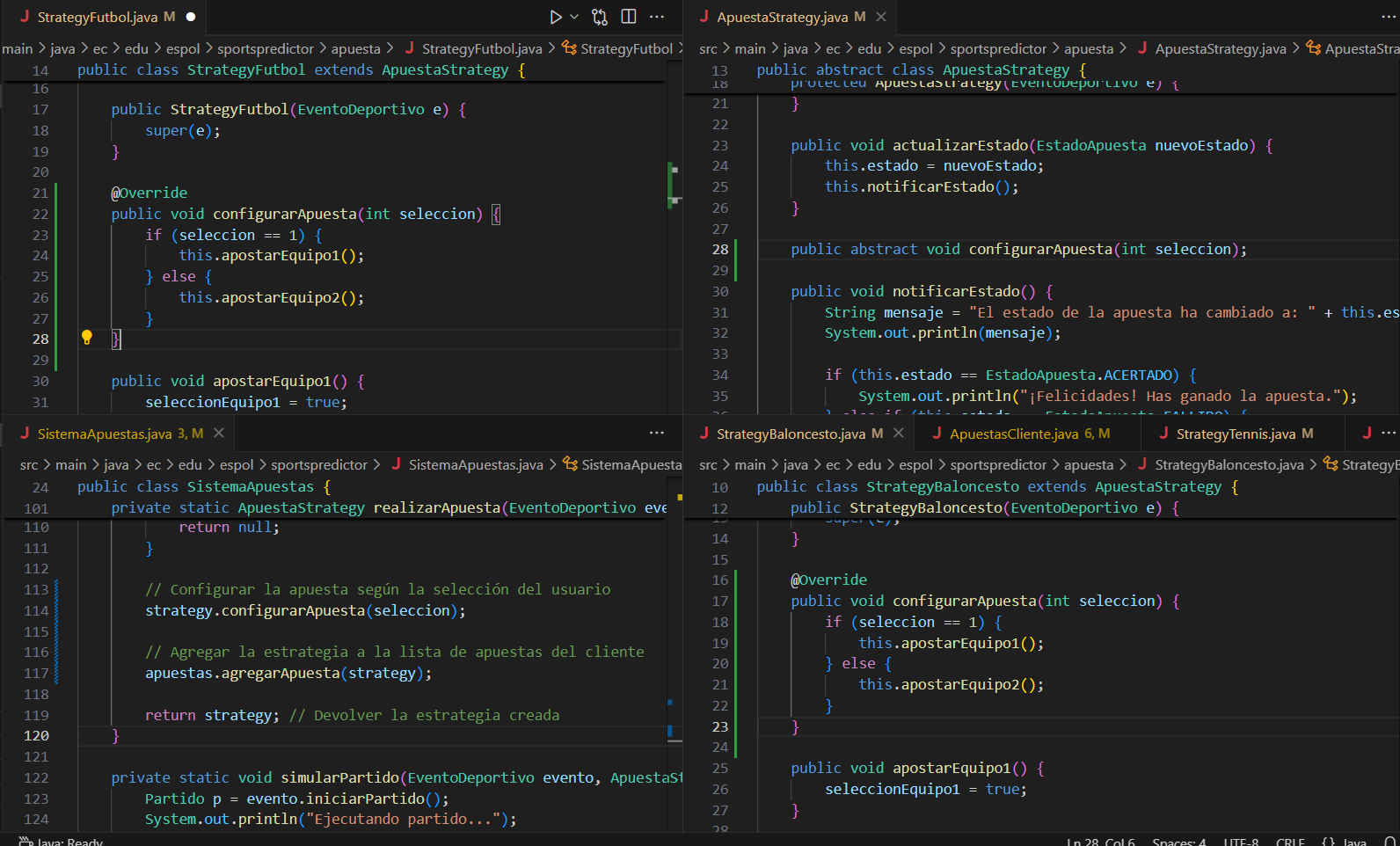
Uso en simularPartido: La estrategia se pasa como parámetro y actualiza el estado de la apuesta según el resultado.

Beneficios

Flujo coherente: Registro, eventos, apuesta y simulación en orden lógico.  
 Código mantenible y claro: Métodos con responsabilidades definidas.  
 Facilidad de pruebas: Métodos probables de forma aislada.

6. Inyección de Dependencias

Esta técnica consiste en pasar las dependencias de una clase como parámetros, en lugar de que la clase las cree o gestione directamente. Esto reduce el acoplamiento y hace que la clase sea más flexible y fácil de mantener.



*Reducción del acoplamiento:* ApuestasCliente ya no depende de ApuestaFactory ni de EventoDeportivo. Solo recibe una estrategia de apuesta y la agrega a su lista. Esto hace que ApuestasCliente sea más independiente y menos propensa a cambios si la lógica de creación de estrategias cambia.

*Mayor flexibilidad:* Ahora es posible crear estrategias de apuestas de diferentes maneras (no solo a través de ApuestaFactory) y pasarlas a ApuestasCliente. Por ejemplo, podrías crear estrategias personalizadas o mock objects para pruebas unitarias.

*Mejor separación de responsabilidades:* La responsabilidad de crear estrategias de apuestas se delega a otra parte del sistema (como SistemaApuestas), mientras que ApuestasCliente se enfoca únicamente en gestionar la lista de apuestas.

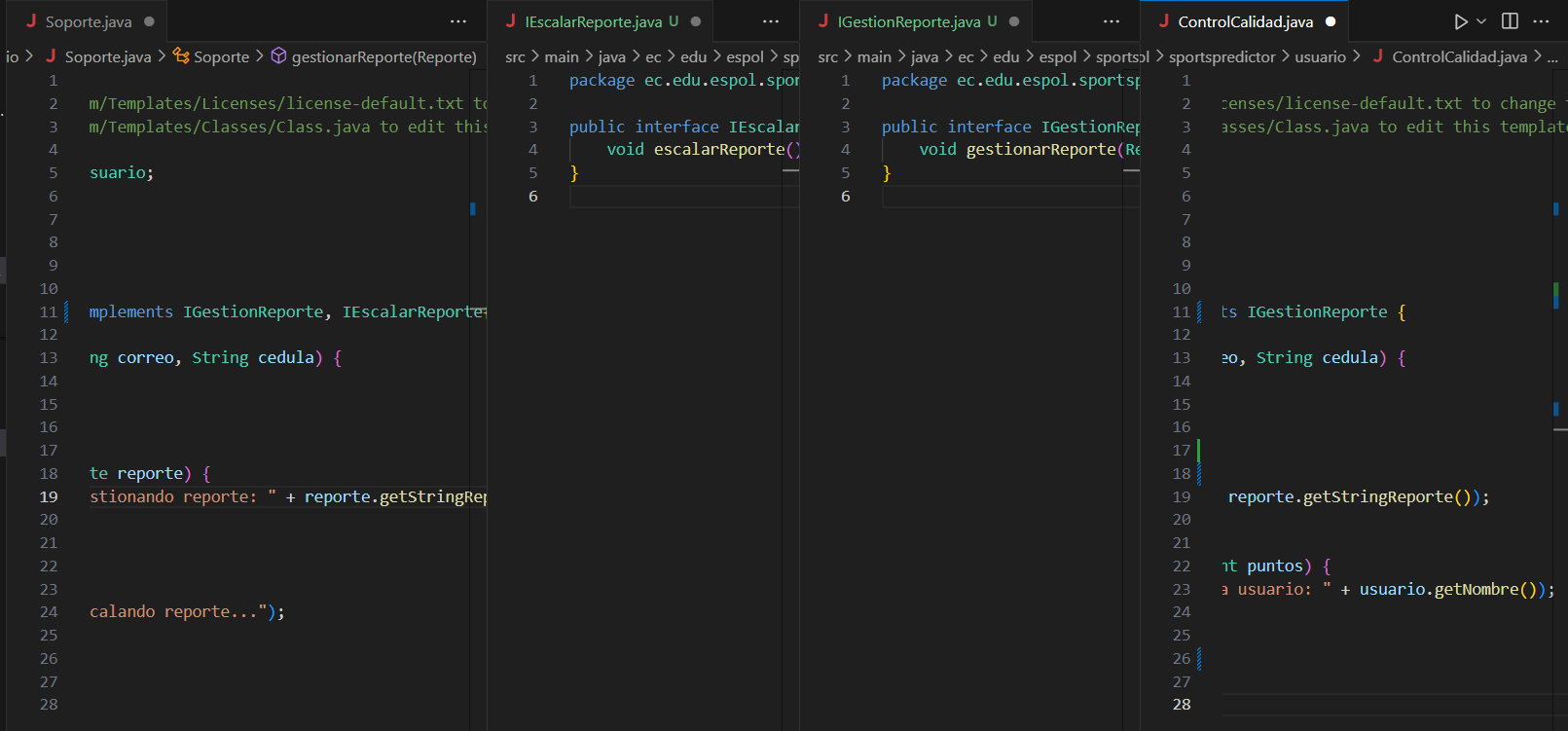
Eliminación del Casting: Ya no necesitamos hacer casting a tipos específicos de estrategias (StrategyFutbol, StrategyTennis, etc.), lo que hace que el código sea más seguro y menos propenso a errores.

Polimorfismo: El método configurarApuesta se comporta de manera diferente según el tipo de estrategia, gracias al polimorfismo.

Código más limpio: El método realizarApuesta es más simple y no necesita conocer los detalles de cada tipo de estrategia.

7. Dividir la Interfaz IControlIncidente

La idea es dividir la interfaz IControlIncidente en interfaces más pequeñas y específicas, de modo que cada clase (ControlCalidad y Soporte) implemente solo las interfaces que realmente necesita. Esto sigue el Principio de Segregación de Interfaces (ISP) y mejora la cohesión del código.



Beneficios de esta Refactorización:

*Principio de Segregación de Interfaces (ISP)*: Ahora las clases solo implementan las interfaces que necesitan. ControlCalidad no está obligada a implementar escalarReporte(), lo que elimina la carga de implementar métodos innecesarios.

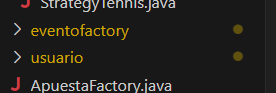
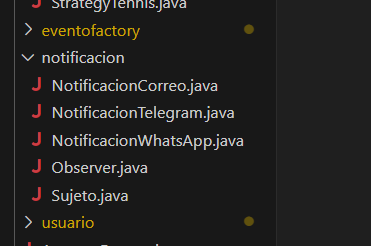
*Mayor Cohesión*: Cada interfaz tiene una responsabilidad única y clara. IGestionReporte se encarga de gestionar reportes, mientras que IEscalarReporte se encarga de escalarlos.

*Flexibilidad*: Si en el futuro surge una nueva clase que solo necesite gestionar reportes pero no escalarlos, simplemente puede implementar IGestionReporte sin verse obligada a implementar IEscalarReporte.

*Mantenibilidad*: El código es más fácil de mantener y entender, ya que las responsabilidades están claramente separadas.

8. Eliminar el Dead Code

Las clases NotificacionCorreo, NotificacionTelegram, y NotificacionWhatsApp pueden considerarse dead code porque no están siendo utilizadas en el flujo principal del programa



*No se Instancian:* En el archivo SistemaApuestas.java (el main), no hay ninguna instancia de las clases NotificacionCorreo, NotificacionTelegram, o NotificacionWhatsApp. Estas clases no se están utilizando para enviar notificaciones a los usuarios.

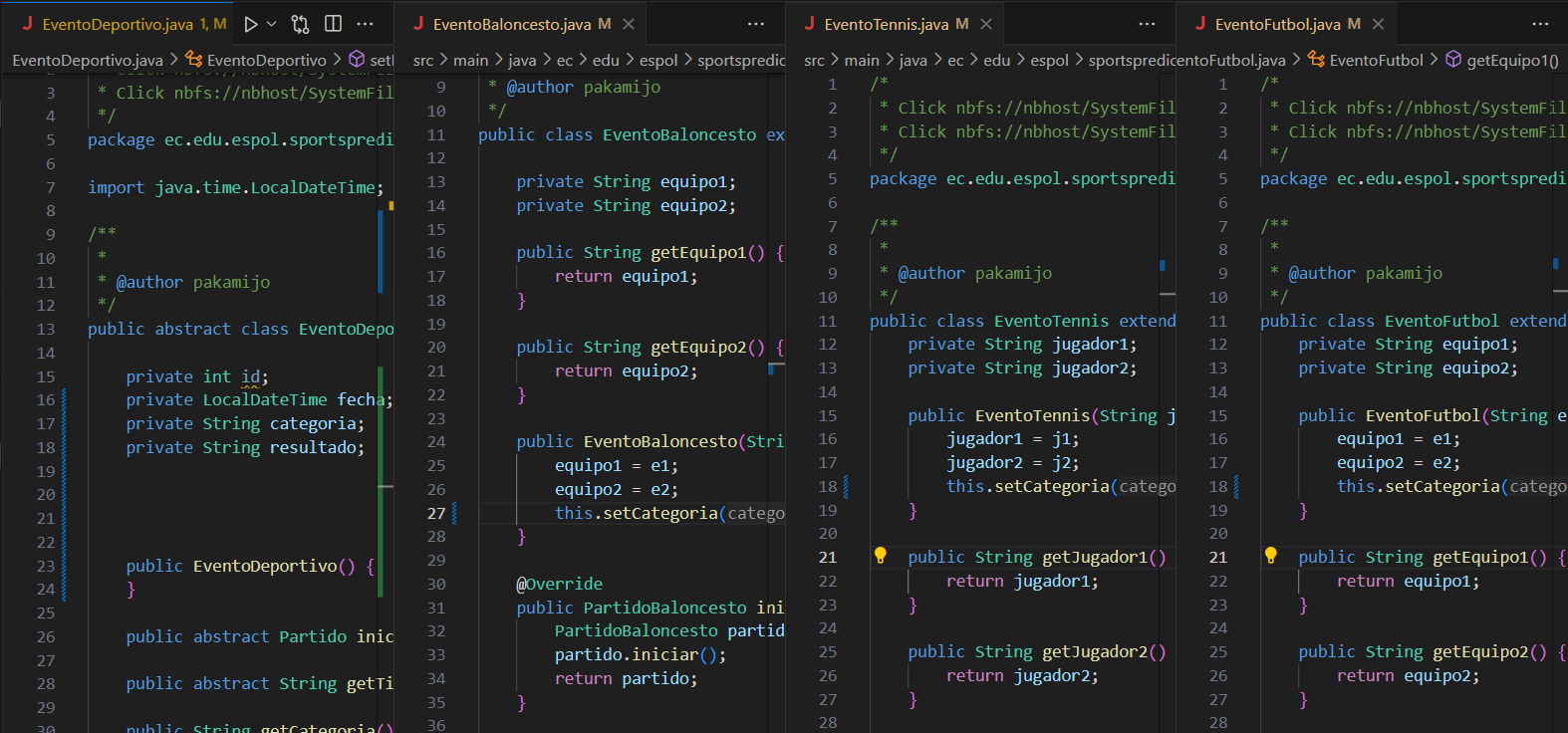
*No se Registran como Observadores:* Aunque estas clases implementan la interfaz Observer, no hay un Sujeto (observable) que las registre. En el patrón Observer, los observadores deben registrarse en un sujeto para recibir notificaciones. Sin este registro, las clases no tienen ningún propósito funcional.

*No se Notifica a los Observadores*: Aunque existe la interfaz Sujeto, no hay una implementación concreta de esta interfaz en el código proporcionado. Por lo tanto, no hay un mecanismo para notificar a los observadores (como NotificacionCorreo, NotificacionTelegram, o NotificacionWhatsApp).

*No se Integran en el Flujo Principal:* En el flujo principal del programa (registro de cliente, selección de evento, realización de apuesta, simulación de partido), no hay ninguna llamada a estas clases de notificación. Esto indica que no están siendo utilizadas en el sistema.

9. Falta de Encapsulamiento

La falta de encapsulamiento en la clase EventoDeportivo es un problema que puede llevar a inconsistencias y dificultades de mantenimiento. Al refactorizar el código para hacer los atributos privados y proporcionar métodos públicos para acceder a ellos, se mejora la robustez, seguridad y mantenibilidad del código. Esta es una práctica esencial en el desarrollo de software orientado a objetos.



**Beneficios de la Refactorización**

*Control sobre el Acceso a los Atributos:* Ahora los atributos categoria y resultado son privados, y solo se pueden acceder a través de los métodos getCategoria() y getResultado(). Si es necesario modificar estos atributos, se debe usar el método setResultado(), donde se pueden agregar validaciones.

*Mantenimiento más Fácil:* Si en el futuro necesitas cambiar la lógica de cómo se almacena o calcula el resultado, solo necesitas modificar los métodos getResultado() y setResultado(), sin afectar otras clases.

*Mayor Seguridad:* Al evitar el acceso directo a los atributos, reduces el riesgo de que otras clases modifiquen los datos de manera incorrecta o inconsistente.

*Cumplimiento del Principio de Ocultación de Información:* Los detalles internos de la clase EventoDeportivo están ocultos, y solo se exponen los métodos necesarios para interactuar con ella.

10. Variables con Nombres No Descriptivos

Al mejorar los nombres de las variables, el código se vuelve más legible y mantenible. Los nombres descriptivos ayudan a entender rápidamente el propósito de cada variable, lo que facilita la colaboración en equipo y la detección de errores.

