Informe de Análisis Dinámico y Estático

Perilla Q. Laura Natalia

Botero G. Santiago

Ingeniería de Sistemas, Escuela Colombiana de Ingeniería Julio Garavito

POOB: Programación Orientada a Objetos

Rocha D. Santiago.

Noviembre 03, 2024

Tabla de Contenido

Abstract

The succeeding report has the objective of evaluating the efficiency and functionalities of the initial project developed for the Object-Oriented Programming course at the Escuela Colombiana de Ingeniería Julio Garavito in the 2024-2 semester. The project draws inspiration from problem F - Tilting Tiles, presented by the ICPC International Collegiate Programming Contest of 2023 held in Luxor, Egypt. The analysis conducted includes both dynamic and static code assessments following the principles of: Code coverage, Java rules, and PMD guidelines.

Resumen

El presente informe tiene como objetivo evaluar la eficiencia y funcionalidades del proyecto inicial desarrollado para el curso de Programación Orientada a Objetos en la Escuela Colombiana de Ingeniería Julio Garavito en el semestre 2024-2. El proyecto se vio influido por el problema F - Tilting Tiles, presentado por la ICP International Collegiate Programming Contest del 2023 que se llevó a cabo en Luxor, Egipto. El análisis dirigido cubre evaluaciones dinámicas y estáticas siguiendo los principios de: Cobertura de código, reglas de Java, y los estándares PMD.

Metodología

El desarrollo del proyecto se llevó a cabo en cuatro ciclos utilizando el entorno integrado BlueJ, diseñado especialmente para principiantes:

- Primer Ciclo: El objetivo principal fue la creación de las clases fundamentales, la interfaz gráfica y las funcionalidades básicas del proyecto. Se incluyeron también características adicionales, como la implementación de pegante en las fichas del tablero.
- Segundo Ciclo: Este ciclo se centró en la ampliación del proyecto mediante la incorporación de ladeos y huecos en el tablero.
- Tercer Ciclo: En esta fase se abordó la resolución del problema planteado por la ICPC, simulando resultados que incluían los ladeos introducidos en el ciclo anterior.
- Cuarto Ciclo: El último ciclo tuvo como objetivo la extensibilidad de los tipos de fichas y los pegantes, modificando los comportamientos asociados a cada tipo de ficha.

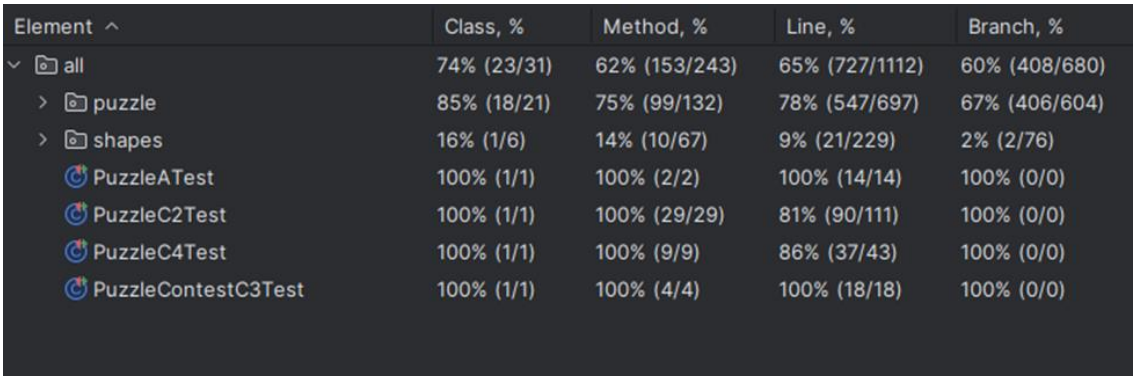
Durante el desarrollo, se siguieron los principios de SOLID y se aplicaron prácticas de programación extrema (XP) para asegurar un código de calidad y un proceso de desarrollo eficiente.

En el marco de cada ciclo se propusieron pruebas de unidad JUnit en su versión 4.8, donde se verificó la funcionalidad de cada método diseñado y el manejo de excepciones. Estas verificaciones fueron pertinentes para el desarrollo del proyecto, ya que permitieron la detección de errores lógicos. Para evaluar los criterios de calidad, funcionalidad y extensibilidad abordados en las pruebas de unidad, se realizó la migración a un entorno de desarrollo integrado diferente, con el fin de verificar la cobertura de las pruebas. Se utilizó IntelliJ IDEA como la IDE preferida para esta tarea.

Análisis Dinámico

Figura 1

Porcentajes de cobertura por paquete.



Element	Class, %	Method, %	Line, %	Branch, %
all	74% (23/31)	62% (153/243)	65% (727/1112)	60% (408/680)
puzzle	85% (18/21)	75% (99/132)	78% (547/697)	67% (406/604)
shapes	16% (1/6)	14% (10/67)	9% (21/229)	2% (2/76)
PuzzleATest	100% (1/1)	100% (2/2)	100% (14/14)	100% (0/0)
PuzzleC2Test	100% (1/1)	100% (29/29)	81% (90/111)	100% (0/0)
PuzzleC4Test	100% (1/1)	100% (9/9)	86% (37/43)	100% (0/0)
PuzzleContestC3Test	100% (1/1)	100% (4/4)	100% (18/18)	100% (0/0)

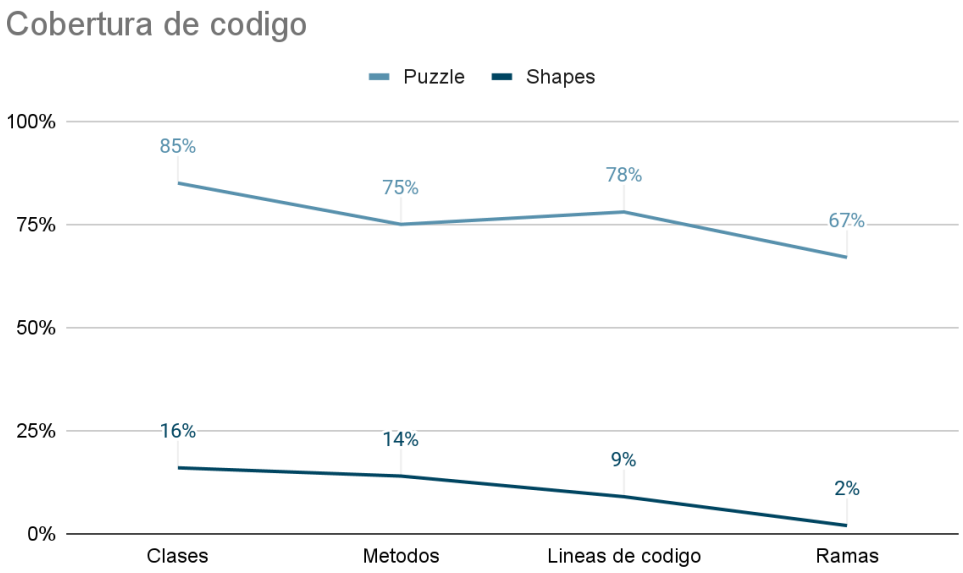
Nota. La captura indica los porcentajes de usabilidad del código de acuerdo a las pruebas de unidad propuesta.

IntelliJ IDEA, permite ejecutar pruebas con cobertura de código. Esto significa que se puede medir qué tan bien están cubiertas las distintas partes del código por las pruebas automatizadas. En el caso del paquete "Puzzle", se obtuvieron los siguientes porcentajes de cobertura:

- 85% de cobertura en las clases: Esto indica que el 85% de las clases del paquete tienen al menos una prueba que las evalúa.
- 75% de cobertura en los métodos: Este porcentaje refleja que el 75% de los métodos están siendo probados por las pruebas automatizadas.
- 78% de cobertura de líneas de código: Esto significa que el 78% de las líneas de código del paquete están cubiertas por pruebas.
- 67% de cobertura de ramas: Este porcentaje se refiere a que el 67% de las posibles rutas de ejecución en el código (debido a condiciones como "if" y "else") están siendo evaluadas por pruebas.

Tabla 1

Cobertura de código por paquete

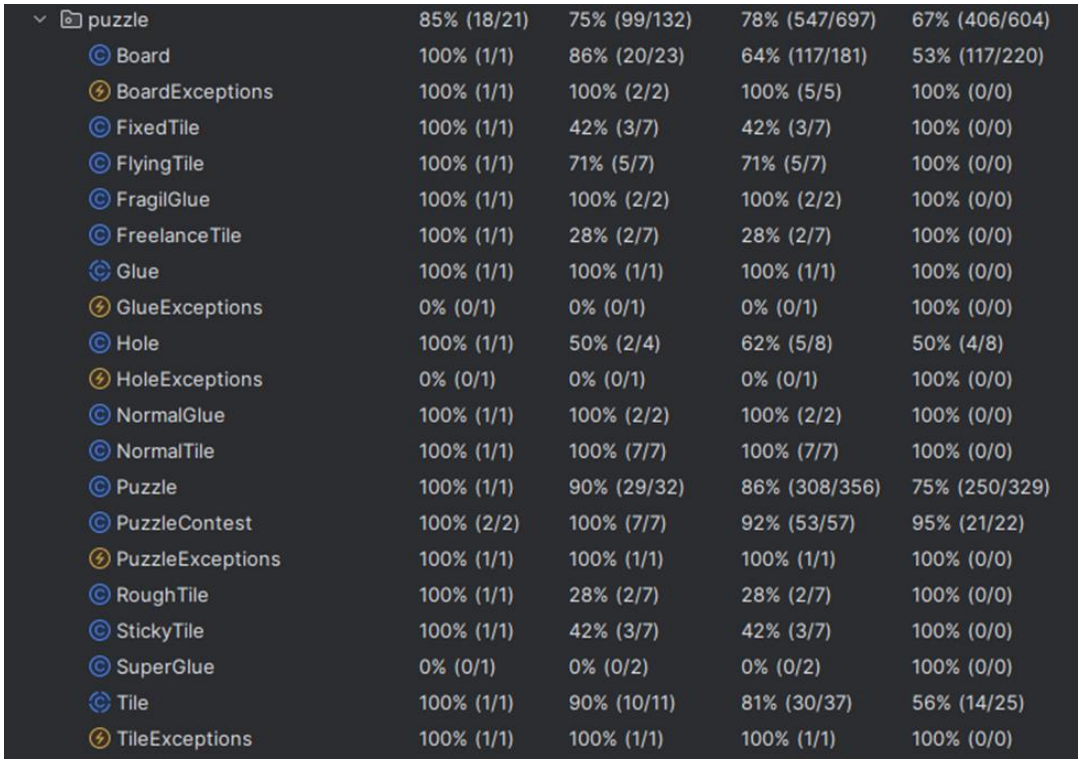


Nota. El gráfico de línea compara los porcentajes de usabilidad por componente.

Estos índices son indicativos de la confiabilidad del código, ya que demuestran la robustez del programa desarrollado. En el caso del paquete Shapes, los porcentajes se pueden hacer caso omiso, dado que solo se emplearon las clases Rectangle y Canvas, junto con algunos métodos específicos que eran fundamentales para los cimientos del proyecto; estos métodos eran los únicos realmente necesarios. Por otro lado, el paquete Puzzle muestra una menor tendencia a errores, según los resultados de las pruebas realizadas. En promedio, los cuatro indicadores alcanzan un promedio de 76%, superando ligeramente el mínimo requerido del 75%.

Figura 2

Porcentajes de cobertura por clase en el paquete Puzzle.



puzzle	85% (18/21)	75% (99/132)	78% (547/697)	67% (406/604)
Board	100% (1/1)	86% (20/23)	64% (117/181)	53% (117/220)
BoardExceptions	100% (1/1)	100% (2/2)	100% (5/5)	100% (0/0)
FixedTile	100% (1/1)	42% (3/7)	42% (3/7)	100% (0/0)
FlyingTile	100% (1/1)	71% (5/7)	71% (5/7)	100% (0/0)
FragilGlue	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)
FreelanceTile	100% (1/1)	28% (2/7)	28% (2/7)	100% (0/0)
Glue	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
GlueExceptions	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
Hole	100% (1/1)	50% (2/4)	62% (5/8)	50% (4/8)
HoleExceptions	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
NormalGlue	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)
NormalTile	100% (1/1)	100% (7/7)	100% (7/7)	100% (0/0)
Puzzle	100% (1/1)	90% (29/32)	86% (308/356)	75% (250/329)
PuzzleContest	100% (2/2)	100% (7/7)	92% (53/57)	95% (21/22)
PuzzleExceptions	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
RoughTile	100% (1/1)	28% (2/7)	28% (2/7)	100% (0/0)
StickyTile	100% (1/1)	42% (3/7)	42% (3/7)	100% (0/0)
SuperGlue	0% (0/1)	0% (0/2)	0% (0/2)	100% (0/0)
Tile	100% (1/1)	90% (10/11)	81% (30/37)	56% (14/25)
TileExceptions	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)

Nota. La captura indica los porcentajes de usabilidad de cada clase de acuerdo a las pruebas de unidad propuestas.

La evaluación de la cobertura de pruebas se realiza mediante un sistema de codificación de colores:

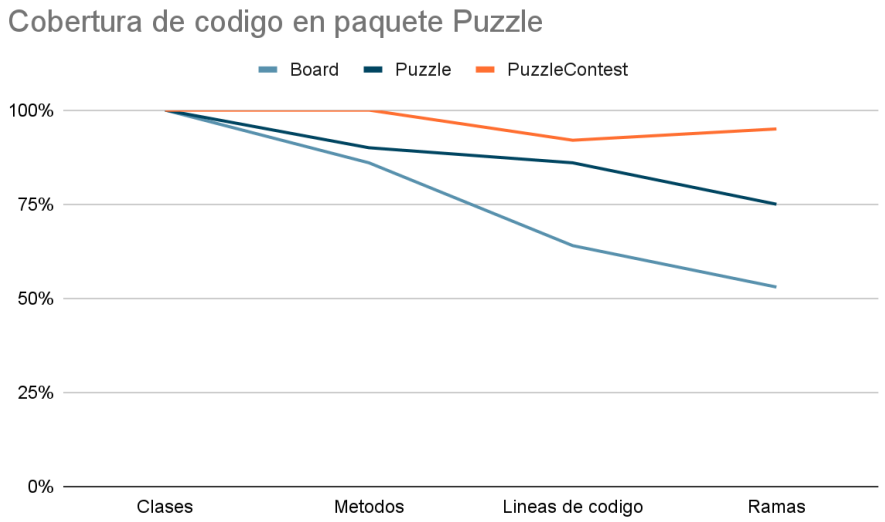
- Verde: Líneas completamente cubiertas por pruebas.
- Amarillo: Líneas parcialmente cubiertas.
- Rojo: Líneas no cubiertas o que no cumplieron con las expectativas.

Tras analizar el código, se concluyó que las pruebas que se consideran parcialmente cubiertas o no cubiertas no evalúan adecuadamente el lanzamiento de excepciones. Esto se debe a que muchas de estas pruebas asumen el correcto funcionamiento del programa. Además, se observó que el constructor general solo gestiona fichas de tipo normal, lo que limita la cobertura

en métodos como exchange() de la clase Puzzle, ya que no se contempla el uso de fichas de diferentes tipos. Por último, se constató que la clase SuperGlue no fue probada, dado que no se encuentra implementada en el código actual, lo que se atribuye a dificultades en su comprensión.

Tabla 2

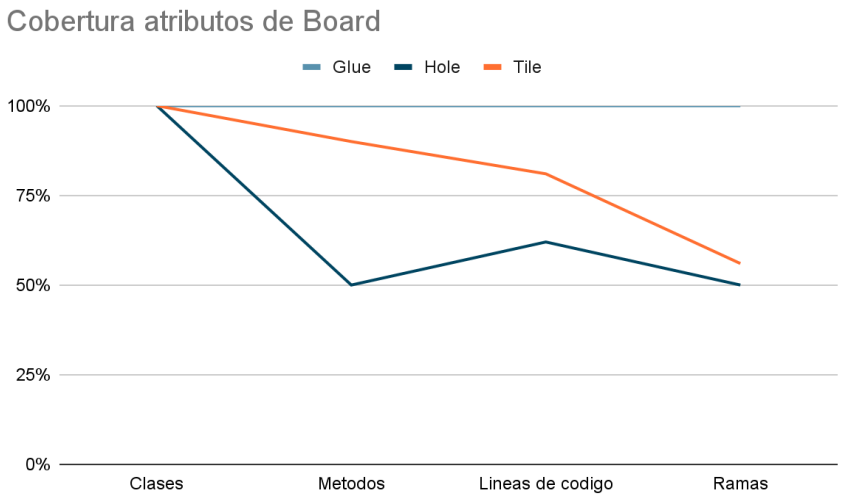
Cobertura del código en el paquete Puzzle.



Nota. El gráfico de línea compara los porcentajes de usabilidad de las clases fundamentales del proyecto.

Tabla 3

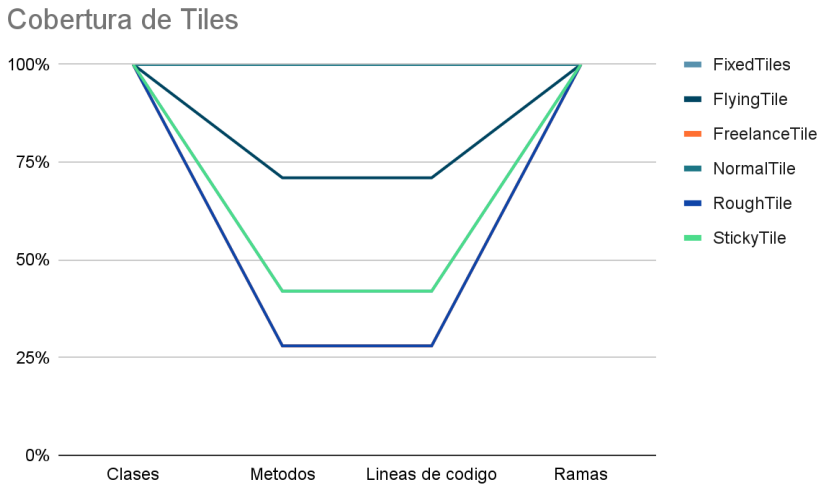
Cobertura de atributos de la clase Board.



Nota. El gráfico de línea compara los porcentajes de usabilidad de las clases que componen el tablero.

Tabla 4

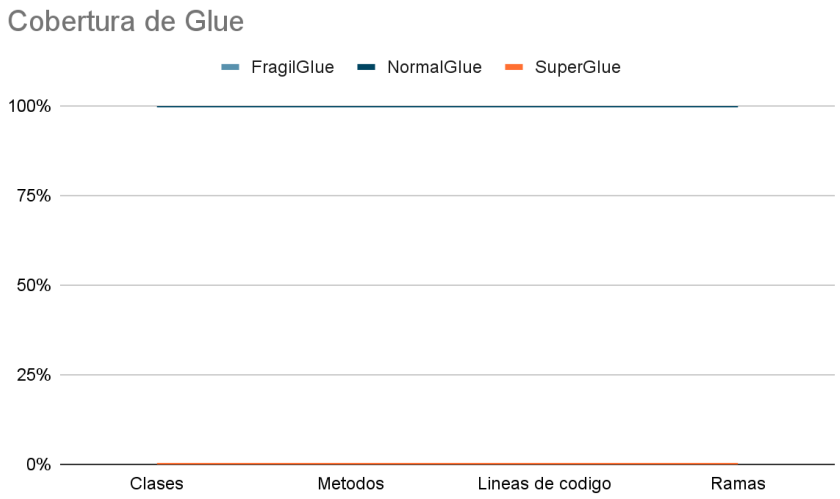
Cobertura de la clase Tile.



Nota. El gráfico de línea compara los porcentajes de usabilidad de las baldosas heredadas de la clase Tile.

Tabla 5

Cobertura de la clase Glue.



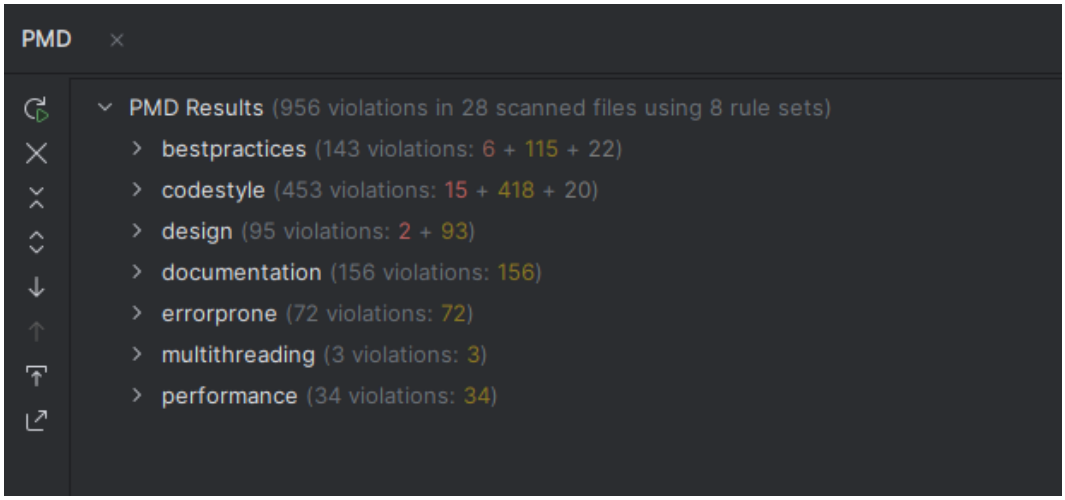
Nota. El gráfico de línea compara los porcentajes de usabilidad de los tipos de pegantes heredados de la clase Glue.

Analisis Estatico

Para optimizar la experiencia del usuario, JetBrains Marketplace es una plataforma ofrecida por JetBrains que permite a los desarrolladores publicar plugins. Estos plugins pueden ser descargados e integrados en entornos de desarrollo como PyCharm, IntelliJ IDEA, DataGrip o CLion. El desarrollador Amit Dev creó el plugin PMD, que permite ejecutar rulesets tanto predefinidos como personalizados.

Figura 3

Resultados PMD según las reglas Java.



Nota. La captura indica las violaciones de acuerdo con los rulesets definidos en quickstart.xml para todos los paquetes del proyecto.

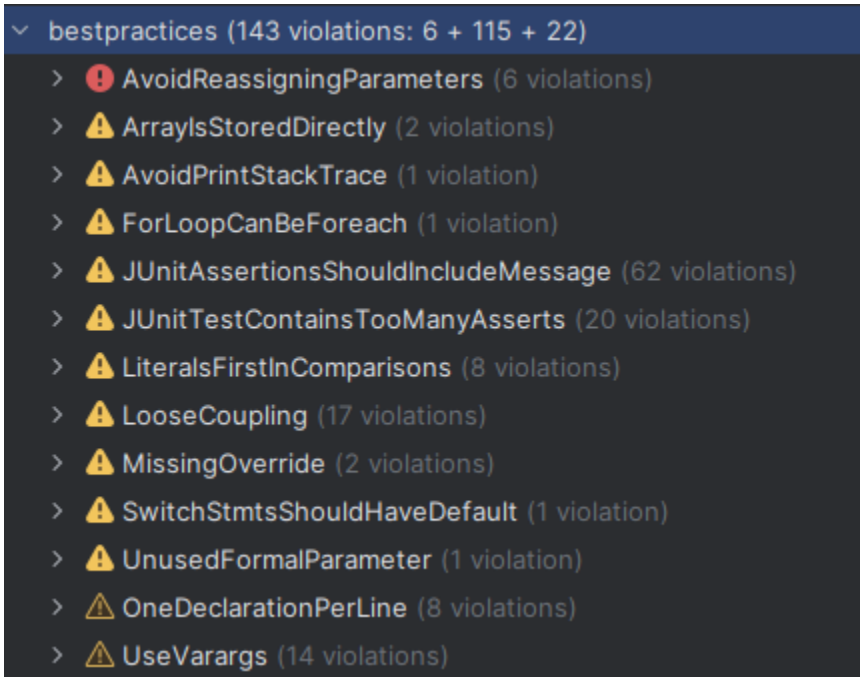
Los resultados generados por el analizador de código PMD evalúan si el código desarrollado cumple con los estándares establecidos, señalando las violaciones a través de un sistema de colores: el rojo indica errores, mientras que el amarillo representa advertencias. En la captura de pantalla proporcionada, se identificaron un total de 956 violaciones en 28 archivos del proyecto, lo que incluye los paquetes de test, puzzle y shapes.

Bestpractices tiene como objetivo mejorar las prácticas de codificación, promoviendo la eliminación de malas prácticas, optimizando la legibilidad y mantenibilidad del código. Las violaciones obtenidas fueron 143.

A continuación, se explicarán con mayor detalle:

Figura 4

Violaciones obtenidas en bestpractices.



Con respecto a AvoidReassigningParameters, surgen 6 violaciones, todas pertenecen al paquete Shapes, en específico estas violaciones son del mismo tipo y aparecen en las clases Circle, Rectangle y Triangle, y es los metodos moveHorizontal() y moveVertical(), aquí se nos informa que es una mala práctica reasignar los valores de los parámetros, lo mejor sería asignar nuevos valores a variables en donde estas no tengan el mismo nombre de las variables que entran como parámetros.

En cuanto a ArrayIsStoredDirectly, surgen dos violaciones en el paquete Puzzle, una de ellas aparece en la clase PuzzleContest, en el método solve(), la otra violación aparece en la clase

auxiliar de PuzzleContest, en el método StateWithMoves(), el problema en los dos métodos es que se están asignando arreglos a variables sin que estos arreglos sean clonados antes de asignarles a una variable, esto puede llegar a ser un problema potencial ya que si el usuario modifica su arreglo original después de pasar el valor, el cambio también se refleja en el objeto en el que almacenamos esa referencia.

AvoidPrintStackTrace presenta una violacion en el paquete Puzzle, específicamente en el método simulate() de la clase PuzzleContest, en este caso se nos recomienda evitar el uso de printStackTrace() en el método y, en su lugar, utilizar un logger para manejar los errores de forma más controlada.

ForLoopCanBeForeach también presenta una sola violación, ésta se presenta en el paquete Shapes, específicamente en la clase Canvas, cuando está implementa el método redraw(), se nos da la recomendación de reemplazar los bucles for tradicionales por bucles foreach también conocidos como "bucles mejorados" cuando sea seguro hacerlo, ya que esto no solo simplifica el código, sino que también mejora su legibilidad.

JUnitAssertionsShouldIncludeMessage presenta 62 violaciones, todas estas pertenecen al paquete Test, a las clases PuzzleATest, PuzzleC2Test, PuzzleC4Test y PuzzleContestC3Test, estas violaciones se presentan debido a que en la mayoría de las pruebas JUnit realizadas no se incluye un mensaje informativo, por lo que se nos recomienda incluir mensajes informativos en ellas con el fin de que sea más fácil identificar problemas cuando una prueba falla.

Por otro lado, JUnitTestContainsTooManyAsserts obtuvo 20 violaciones en el paquete Test, estas violaciones pertenecen a las clases PuzzleC2Test y PuzzleC4Test, en la mayoría de las pruebas de estas dos clases se presenta más de un Assert por prueba y aquí se nos informa que las pruebas unitarias no deben contener más de una afirmación (assert). Esto se sugiere porque tener

demasiadas afirmaciones en una sola prueba puede complicar la verificación de la corrección de esta.

LiteralsFirstInComparisons arroja 8 violaciones, donde todas pertenecen al paquete Shapes, específicamente a la clase Canvas, en el método setForegroundColor() se están realizando varias comparaciones entre valores y variables que entran como parámetros, se nos aclara que cuando se van a realizar comparaciones, lo que va en el paréntesis es la variable que ingresó como parámetro y se desea comparar, es decir, lo ideal es "2".equals(x) y no x.equals("2") qué es lo que se puso en todas las comparaciones que se realizaron en el método.

LooseCoupling arroja 17 violaciones, estas pertenecen a los paquetes Puzzle y Shapes en donde se involucra Canvas, con respecto a Puzzle, las clases involucradas son PuzzleContest, Board y Puzzle, en estas clases anteriormente nombradas, se declaran contenedores concretos como HashSet, debido a esto se nos recomienda evitar el uso de estos y reemplazarlo por interfaces, como lo son Set o List ya que nos puede facilitar el cambio a diferentes implementaciones en el futuro si los requisitos cambian.

MissingOverride obtuvo 2 violaciones, una se encuentra en el paquete Shapes, en la clase auxiliar de Canvas , en el método paint(), la otra violación se encuentra en el paquete Puzzle, en la clase NormalGlue, en el método canTilt(), los métodos nombrados anteriormente son sobrescritos, pero no se escribió la anotación @Override arriba de la firma del método, se nos recomienda ponerlo ya que ayuda al compilador a verificar que efectivamente se está sobrescribiendo un método existente.

SwitchStmtsShouldHaveDefault obtuvo una violación en la clase Puzzle, en el método addGlue() ya que en este se encuentra una sentencia Switch, donde no se manejó el caso default que es cuando ninguno de los casos anteriores se cumple, por lo que se recomienda que las

sentencias switch deben ser exhaustivas, lo que significa que deben manejar todos los posibles casos.

UnusedFormalParameter obtuvo una violación en la clase puzzle, en el método moveHorizontal() ya que nunca se está utilizando el parámetro row, por lo que se nos hace la recomendación de evitar poner parámetros que no se van a usar en el cuerpo del método.

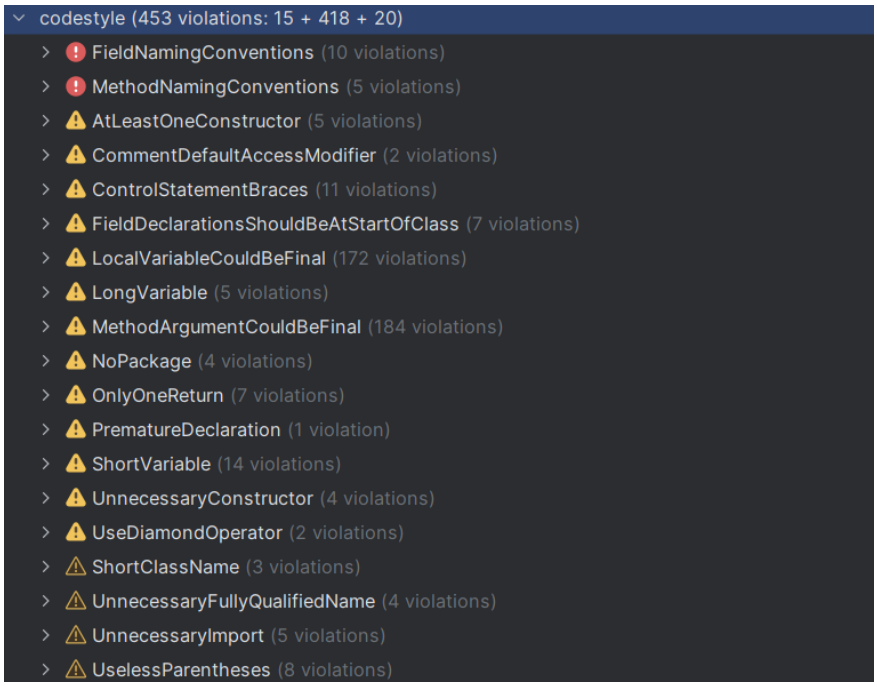
OneDeclarationPerLine contiene 8 violaciones, las cuales pertenecen a la clase Puzzle, en donde las clases que declaran variables en este paquete están declarando variables, pero en una misma línea, no una declaración por línea, razón por la cual se nos recomienda usar una línea para cada declaración de variable, ya que esto mejora la legibilidad del código.

Finalmente, UseVarargs contiene 14 violaciones, todas en el paquete Puzzle, y las violaciones se encuentran en las clases Board, Puzzle y PuzzleContest, se nos recomienda considerar el uso de varargs (argumentos variables) para métodos o constructores que toman un arreglo como último parámetro.

Los parámetros establecidos por el **codestyle** definen un estándar específico que rige las conductas que deben ser mantenidas o evitadas en la programación. El analizador de código obtuvo las 453 violaciones siguientes:

Figura 5

Violaciones obtenidas en codestyle.



FieldNamingConventions reportó 10 violaciones en las clases BoardException, Triangle y Rectangle. El propósito de esta convención es utilizar camel case para nombrar las variables. La razón por la que BoardException generó un error se debe a que no se declararon las constantes como finales; por lo tanto, el analizador no las identificó como constantes, lo que resultó en el error de nombramiento.

MethodNamingConventions reportó 5 violaciones en las clases PuzzleATest y Puzzle. El propósito de esta convención es utilizar camel case para nombrar los métodos. Los métodos que generaron el error comenzaron con mayúscula.

AtLeastOneConstructor reportó 5 violaciones en las clases PuzzleATest, PuzzleContestC3Test, PuzzleC2Test y PuzzleC4Test. El propósito de esta convención es asegurar que cada clase tenga, como mínimo, un constructor.

CommentDefaultAccessModifier reportó 2 violaciones en la clase PuzzleContest. El propósito de esta convención es incluir comentarios o anotaciones al inicio de cada declaración de clase o método que utilice un modificador de acceso default, con el fin de evitar confusiones. Por ejemplo, si existe un atributo como char[][] state;, al tener un modificador de acceso por defecto, debería declararse de la siguiente manera: /* default */ char[][] state;.

ControlStatementBraces reportó 11 violaciones, todas ellas ubicadas en las clases Rectangle y Canvas del paquete Shapes. Estas declaraciones no corresponden con los objetivos del proyecto.

FieldDeclarationsShouldBeAtStartOfClass reporto 7 violaciones, las cuales se encuentran en la clase Canvas. No obstante, se puede hacer caso omiso a estas declaraciones.

LocalVariableCouldBeFinal reportó 172 violaciones en las siguientes clases: PuzzleATest, Triangle, Circle, Rectangle, PuzzleContestC3Test, PuzzleC4Test, PuzzleContest, PuzzleC2Test, Canvas, Board y Puzzle. El propósito de esta convención es declarar variables como finales si estas solo fueron asignadas una única vez.

LongVariable reportó 5 violaciones en PuzzleContest, PuzzleC2Test y Puzzle. Esta convención tiene como objetivo prevenir el uso de nombres extensos para variables locales o argumentos, promoviendo así la claridad y la legibilidad del código.

MethodArgumentCouldBeFinal reporto 184 violaciones en las siguientes clases: HoleExceptions, BoardExceptions, PuzzleExceptions, FlyingTile, NormalTile, FixedTile, FreelanceTile, TileExceptions, StickyTile, RoughTile, Hole, Tile, Triangle, Circle, Rectangle,

GlueExceptions, PuzzleContest, StateWithMoves, Canvas, ShapeDescription, Board y Puzzle. El objetivo de esta convención es asignar la palabra clave "final" a aquellos métodos que no requieren ser heredados, con el fin de prevenir sobreescrituras y mejorar la seguridad del programa.

NoPackage reporto 4 violaciones en las clases PuzzleATest, PuzzleC2Test, PuzzleContesC3Test y PuzzleC4Test. El propósito de esta convención es asegurar que cada clase esté asignada a un paquete. Sin embargo, estas clases fueron extraídas de sus paquetes para permitir que IntelliJ ejecute las pruebas unitarias en el directorio establecido para pruebas, de acuerdo con la estructura del proyecto en la aplicación.

OnlyOneReturn reportó 7 violaciones en Tile, Rectangle, PuzzleContest, Board y Puzzle. Los métodos que retornan valores deberían tener únicamente una salida. En los métodos de las clases con violaciones, se encontraron múltiples salidas debido a condiciones en estructuras if o for, lo que no es una práctica adecuada.

PrematureDeclaration reportó 1 violación en Puzzle.glueMovablePositions(). Esta regla no tiene como objetivo optimizar el código, sino mejorar la legibilidad para el lector. Es preferible evitar la declaración de variables antes de que sean realmente necesarias.

ShortVariable reportó 14 violaciones en las clases Circle, Rectangle, PuzzleContest, PuzzleC2Test, CanvasPane, Board y Puzzle. Se considera una buena práctica utilizar nombres de variables locales que sean más largos y descriptivos, ya que esto mejora la legibilidad del código para el lector.

UnnecessaryConstructor reportó 4 violaciones en las clases Glue, NormalGlue, FragilGlue y SuperGlue. Esta convención tiene como objetivo eliminar constructores innecesarios que no contribuyan a la estructura del código. En este caso, todos los constructores de las clases de

pegamento estaban vacíos, lo que equivale al constructor privado por defecto de Java, y por lo tanto, su implementación resulta innecesaria.

UseDiamondOperator reportó 2 violaciones en la clase Canvas. Dado que esta clase no está relacionada con el propósito del análisis, se puede hacer caso omiso.

ShortClassName reportó 3 violaciones en las clases Glue, Hole, Tile. Esta convención recomienda que los nombres de las clases tengan más de cinco caracteres.

UnnecessaryFullyQualifiedName reporto 4 violaciones en las clases Puzzle para Board.exchange(). Esta convención establece que no es necesario utilizar el nombre completo de las constantes dentro de la misma clase en la que fueron definidas. En este caso, se empleó el nombre completo Board.ENDING_BOARD dentro de la clase Board, donde se definió la constante.

UnnecessaryImport reportó 5 violaciones en las clases Tile, Circle, Rectangle, PuzzleContest. Esta convención establece que las bibliotecas importadas que no se han utilizado o que están duplicadas son innecesarias para la estructura del código.

UselessParentheses reportó 8 violaciones en Board y Puzzle. Esta convención establece que los paréntesis deben utilizarse únicamente para sobrescribir operadores; si eliminar un paréntesis no altera el resultado de la operación, no debería ser incluido.

Design tiene como objetivo entregar un conjunto de reglas las cuales se enfocan en mejorar el diseño y la estructura del código. El analizador de código obtuvo 95 violaciones.

A continuación, se explicarán con mayor detalle esas violaciones

Figura 6

Violaciones obtenidas en design.

Con respecto a ClassWithOnlyPrivateConstructorsShouldBeFinal arroja 2 violaciones, ambas se encuentran en el paqueteShapes, en las clases Canvas y en su clase privada CanvasPane, estas dos clases se crearon como pública y privada respectivamente, pero se nos recomienda marcar las clases con solo constructores privados como final. Esto se debe a que tales clases no pueden ser extendidas desde fuera de su unidad de compilación, lo que significa que no tiene sentido permitir que sean subclasificadas. Al hacer una clase final, se evita la posibilidad de que otros desarrolladores intenten heredar de ella, lo que puede ayudar a prevenir errores y mantener la intención del diseño original.

AvoidCatchingGenericExcepcion arroja 29 violaciones, unas pertenecen al paquete Test, específicamente a sus clases PuzzleC2Test y PuzzleC4Test, otras se encuentran en el paquete

Shapes, en su clase Canvas y por último las otras se encuentran en el paquete Puzzle, en la clase puzzle, lo que sucede es que las pruebas, el método wait(), moveHorizontal() y moveVertical() están capturando excepciones genéricas, como lo son NullPointerException, RuntimeException o Exception en bloques Try/Catch, por lo que se nos recomienda evitar capturar este tipo de excepciones en ese tipo de bloques ya que puede llevar a un manejo ineficaz de los errores y complicar la depuración del código.

CognitiveComplexity arroja 5 violaciones, estas pertenecen a la clase puzzle, específicamente a uno de los constructores, relocateTile(), moveHorizontal(), moveVertical() y tempMatrix(), en donde estas tienen anidamientos muy grandes, por lo tanto se nos advierte que estos métodos tienen una complejidad cognitiva bastante alta, superando el umbral de 15. La complejidad cognitiva mide cuán difícil es para los humanos leer y entender un método, tomando en cuenta el uso de estructuras de control y el nivel de anidamiento.

CollapsibleIfStatements arroja una única violación la cual pertenece a la clase Puzzle, en específico al método glueMovablePositions(), en donde este está realizando una verificación en un bloque if y seguido a esta línea de código, se encuentra otra verificación con un if, por lo que se sugiere que se combinen declaraciones if anidadas cuando sea posible.

CyclomaticComplexity arroja 16 violaciones, todas pertenecen al paquete Puzzle, en específico a las clases Board y Puzzle en donde se encuentran varias métodos, los cuales tienen varios ciclos anidados lo que llega a ser bastante complejo, por lo que los hablan de complejidad ciclomática y que la estamos excediendo. Para reducir la complejidad ciclomática, se nos recomienda dividir el método en varios métodos más pequeños, cada uno manejando una parte específica de la lógica. Esto también facilita la reutilización y el testing de cada componente.

DataClass arroja 3 violaciones, todas ellas en el paquete Puzzle, específicamente en las clases PuzzleExceptions, BoardExceptions, TileExceptions debido a que se nos indica que estas clases están actuando como una Data Class (o clase de datos), que se limita a almacenar datos sin lógica adicional o comportamientos asociados. Este tipo de clases tienden a contener solo campos públicos o métodos de acceso sin operaciones complejas, lo que dificulta el mantenimiento y reduce la cohesión entre los datos y el comportamiento.

ImmutableField arroja 17 violaciones, unas pertenecen al paquete Shapes, en este caso a las clases y las otras pertenecen al paquete Puzzle, las violaciones surgen porque se nos aconseja declarar varias variables como finales si su valor nunca cambia después de la inicialización del objeto.

LawOfDemeter arroja 2 violaciones, de las cuales una pertenece al paquete Shapes, en este caso al método setVisible de la clase Canvas y la otra al paquete Puzzle, en la clase Board, específicamente en el método deleteGlueAfterTilt(), con respecto a estos métodos, el problema se debe a que se está llamando a dos métodos, por medio de una variable que viene desde antes, por lo que nos dicen que esto se relaciona con la Ley de Demeter, que propone reducir el acoplamiento entre clases limitando las dependencias indirectas, es decir, la cantidad de "saltos" o llamadas a métodos en la cadena de objetos.

MutableStaticState arroja 7 violaciones, algunas pertenecen al paquete Shapes y las otras pertenecen al paquete Puzzle, aquí algunas variables son declaradas como estáticas, pero también públicas, por lo tanto, se nos advierte que los campos static públicos o protegidos que no son final, pueden modificarse desde cualquier lugar del programa, lo que puede romper la encapsulación y provocar errores difíciles de rastrear.

SimplifyBooleanExpressions arroja 2 violaciones en la clase Board, específicamente en el método changeTilesVisibility(), debido a que este método está realizando comparaciones booleanas innecesarias por lo que se nos recomienda hacer comparaciones realmente importantes, ya que realizar comparaciones que no aportan nada al resultado puede hacer que el código sea más difícil de leer y entender.

SimplifyBooleanReturns arroja 2 violaciones, una en la clase Tile, en el método hasGlue(), la otra es la clase Board en el método doesThisPositionHaveHole(), estos métodos están usando condicionales if innecesarios, porque se nos recomienda usar solo el resultado de una comparación en lugar de usar un if.

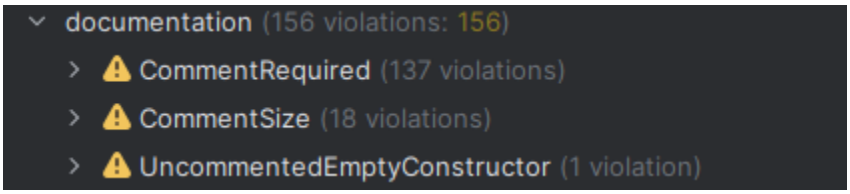
SingularField arroja 2 violaciones, estas pertenecen a la clase PuzzleContest en donde se están declarando variables que perfectamente podrían ser declaradas variables locales, por lo tanto, se nos recomienda que si un campo de clase es asignado antes de ser leído en todos los métodos donde se utiliza, entonces no es necesario almacenarlo como un campo de instancia.

TooManyMethods arroja 7 violaciones, estas pertenecen a los paquetes Shapes, Puzzle o Test, estos métodos contienen clases que a su vez algunas de ellas contienen varios métodos, por lo que llegan a ser demasiado complejas, como lo son las clases Puzzle, Board, Tile, entre otras. Nos recomiendan dividir la clase en componentes más pequeños y manejables, siguiendo el principio de responsabilidad única.

El informe de **Documentation** resalta las estructuras y parámetros que deben seguirse en la documentación del proyecto. Según las PMD, se identificaron 156 violaciones:

Figura 7

Violaciones obtenidas en documentation.



CommentRequired indicó 137 violaciones en las clases Glue, NormalGlue, GlueExceptions, HoleExceptions, FixedTile, FragilGlue, NormalTile, Hole, Board, SuperGlue, PuzzleExceptions, RoughTile, FlyingTile, StickyTile, TileExceptions, FreelanceTile, Tile, PuzzleATest, Triangle, Circle, PuzzleContestC3Test, Rectangle, PuzzleC4Test, PuzzleContest, StateWithMoves, PuzzleC2Test, Board, Canvas, ShapeDescription y Puzzle. Este informe destaca la necesidad de añadir comentarios en métodos o atributos para ciertos elementos del lenguaje.

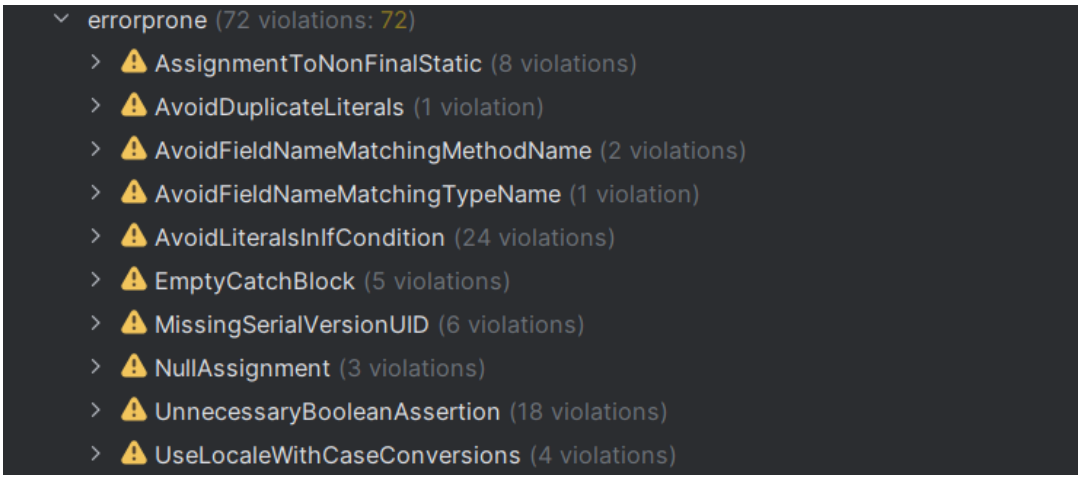
CommentSize indicó 18 violaciones en las clases PuzzleContest, PuzzleC2Test, Canvas y Puzzle. Indica comentarios en líneas demasiado largas saliéndose de los límites especificados.

UncommentedEmptyConstructor indicó una violación por la clase Glue.Glue(). Dado que los constructores de la clase Glue y sus subclases están vacíos, si se requiere mantener estos constructores, se debería incluir un comentario que indique que su implementación vacía es intencional.

Error Prone lanza alerta de diseños que están dañados, extremadamente confusos o propensos a runtime errors. Se identificaron 72 violaciones:

Figura 8

Violaciones obtenidas en errorprone.



AssignmentToNonFinalStatic reporta 8 violaciones en las clases Board y Puzzle. Esta convención lanza una alerta cuando se asignan valores a variables estáticas finales a partir de variables no finales, lo que podría implicar una reducción en la seguridad del código.

AvoidDuplicateLiterals reporta una violación por la clase Puzzle.Puzzle(). Se recomienda evitar la redundancia en el código al contener literales duplicados. Esto se podría mejorar declarando una variable estática para su reutilización, lo que optimizaría la claridad y mantenibilidad del código.

AvoidFieldNameMatchingMethodName reporta 2 violaciones en las clases Rectangle y Puzzle. Se debe evitar el uso de nombres idénticos para variables y métodos, ya que puede generar confusión. En este caso, el atributo colorMap en la clase Puzzle, que almacena un HashMap de colores disponibles, presenta esta situación.

AvoidFieldNameMatchingTypeName reporta una violación en la clase Canvas. Sin embargo, dado el propósito del proyecto, se puede hacer caso omiso a esta indicación.

AvoidLiteralsInIfCondition reporta 24 violaciones en las clases PuzzleContest y Puzzle. Se recomienda evitar el uso de literales directamente en las sentencias condicionales. En su lugar, es preferible declarar estos literales como variables estáticas con nombres descriptivos, lo que mejora la mantenibilidad del código. Alternativamente, se podrían explorar otros enfoques para resolver el problema de manera más eficiente.

EmptyCatchBlock reporta 5 violaciones, una de ellas pertenece al paquete Shapes, específicamente a su clase Canvas, las otras pertenecen al paquete Puzzle, a la clase Puzzle, donde nos indican que se están usando bloques catch vacíos, esto puede llegar a ser un problema importante ya que se están atrapando las excepciones, pero no se realiza ninguna acción con ellas.

MissingSerialVersionUID arroja 6 violaciones, una de ellas pertenece al paquete Shapes, específicamente a la clase auxiliar de Canvas, la cual se llama CanvasPane, las otras pertenecen al paquete Puzzle, específicamente a las clases que contienen excepciones, en donde estas están implementando Serializable, por lo que se nos dice que existe la necesidad de establecer un campo serialVersionUID en las clases que implementan la interfaz Serializable en Java. Este identificador es fundamental para asegurar la compatibilidad durante el proceso de serialización y deserialización.

NullAssignment arroja 3 violaciones en las clases Tile y PuzzleConstest ya que estas están asignando en ciertos métodos null a objetos después de utilizarlos, por lo que se nos informa que es mejor refactorizar el código para eliminar tales asignaciones mejora la claridad y la mantenibilidad de este, lo cual es una buena práctica en programación.

UnnecessaryBooleanAssertion arroja 18 violaciones en las clases PuzzleC2Test y PuzzleC4Test, en donde se informa que se están utilizando asserts innecesarios, como lo es assertTrue(true), por lo que no está aportándole valor a las pruebas.

UseLocaleWithCaseConversions arroja 4 violaciones en las clases Tile, Board y Puzzle, en los métodos addGlue() y addTile(), nos hablan de la importancia de especificar un Locale explícito al realizar conversiones de cadenas a minúsculas o mayúsculas en Java. No hacerlo puede conducir a resultados inesperados debido a las diferencias en las reglas de transformación.

El informe de **Multithreading** alerta sobre comportamientos que pueden provocar problemas de rendimiento al ejecutar múltiples hilos. Se alzaron 3 violaciones:

Figura 9

Violaciones obtenidas en multithreading.

DoNotUseThreads arrojó 2 violaciones en PuzzleContest.simulate() y Canvas.wait(). Según las especificaciones de J2EE, se prohíbe el uso de hilos. Sin embargo, en el método simulate() de la clase PuzzleContest, el uso del Thread.sleep(milliseconds) en un bloque try-catch era necesario para simular el movimiento de las baldosas y visualizar los ladeos del tablero.

NonThreadSafeSingleton indico una violación en la clase Canvas. No obstante, dado que este análisis no es relevante para el propósito del proyecto, se puede hacer caso omiso.

Performance genera alertas sobre líneas de código que podrían afectar la eficiencia y el rendimiento debido a prácticas subóptimas. Las pruebas PMD detectaron un total de 34 violaciones en este sentido:

Figura 10

Violaciones obtenidas en performance.

AvoidInstantiatingObjectsInLoops identificó 31 violaciones en las clases PuzzleContest, Board y Puzzle. Esta convención recomienda evitar la instanciación de nuevos objetos dentro de estructuras de repetición, ya que puede generar sobrecarga al no reutilizar objetos. Sin embargo, en este caso, la creación de nuevos objetos era necesaria para la estructura del proyecto, ya que se requería regresar a estados anteriores del tablero. Para lograrlo, se crean copias de las baldosas mediante un constructor auxiliar, lo que permitía guardar un respaldo del tablero antes de realizar cambios.

UselessStringValueOf indico 3 violaciones en la clase Puzzle. En este caso, la conversión de un char a String utilizando String.valueOf() resultó innecesaria, ya que se estaba realizando una concatenación en los lanzamientos de excepciones, como en el siguiente código:

```
throw new
PuzzleExceptions(PuzzleExceptions.INVALID_COLOR+String.valueOf(ending[i][j]))
;
```

Esta conversión de char a String no aporta valor adicional, ya que la concatenación de un char con una cadena ya lo convierte implícitamente en String.

Conclusiones

A lo largo del análisis realizado, se han identificado un total de 956 violaciones en el código del proyecto. De estas, 23 violaciones son de prioridad alta, lo que indica que representan áreas críticas que deben ser corregidas para asegurar la estabilidad, seguridad y rendimiento del sistema. Adicionalmente, 10 violaciones están relacionadas con el paquete Shapes. A pesar de estos indicadores, solo 13 violaciones restantes asociadas al paquete Puzzle deben ser abordadas para cumplir con los estándares de calidad y el cumplimiento de la meta propuesta. Es importante destacar que, aunque estas violaciones existen, no han afectado de manera significativa el funcionamiento actual del sistema, lo que implica que, en términos generales, el código es funcional, pero podría optimizarse aún más.

En cuanto a la cobertura de pruebas, el proyecto ha alcanzado un 75% de cobertura de código, lo cual es un logro importante. Sin embargo, este nivel de cobertura deja espacio para la mejora. Sería beneficioso realizar más y mejores pruebas unitarias, en particular aquellas que aborden las áreas que presentan violaciones de alta prioridad. Esto no solo mejoraría la calidad del código, sino que también fortalecería la confiabilidad del sistema en escenarios no previstos y garantiza que las futuras modificaciones no impliquen daños al programa.

Al revisar detalladamente cada una de las violaciones, es evidente que, aunque el código actual cumple con los requisitos funcionales, siempre existe espacio para la mejora continua. Las violaciones, aunque no siempre críticas, señalan oportunidades para optimizar el código, mejorar su legibilidad, reducir la complejidad y aumentar la seguridad. La mejora del código no solo se trata de corregir errores, sino de hacerlo de una manera que permita un mantenimiento más sencillo y un mayor rendimiento a largo plazo. Esto incluye refactorizaciones pequeñas que, si bien no

afectan directamente el funcionamiento, pueden tener un impacto significativo en la mantenibilidad y escalabilidad del proyecto.

En conclusión, aunque el proyecto está funcionando adecuadamente, es claro que siempre hay puertas abiertas para mejorar. La corrección de las violaciones, el aumento de la cobertura de pruebas y la implementación de mejores prácticas pueden transformar este proyecto en una base de código más robusta, eficiente y fácil de mantener a largo plazo.