Escuela Colombiana de Ingeniería Julio Garavito



**Análisis estático**

Presentado por:

Camilo Andrés Fernández Diaz

Andrés Jacobo Sepúlveda Sánchez

Asignatura:

Programación orientada a objetos

Grupo:

ISIS POOB-4

Docente:

Juan Sebastián Frasica Galeano

María Irma Diaz Rozo

El propósito de este informe es documentar los resultados del análisis estático realizado sobre el código fuente del proyecto Puzzle utilizando la extensión PMD en el entorno de eclipse. Este análisis se enfoca en identificar posibles problemas de calidad de código y aplicar buenas prácticas recomendadas para el desarrollo en Java.

El análisis se ejecutó sobre todas las clases referentes al proyecto las cuales son:

* Puzzle
* PuzzleC2Test
* PuzzleC2ExtendTests
* PuzzleATest
* PuzzleBTest
* Glue
* Rectangle
* Tilt
* Canvas
* PuzzleContest

**Resumen general de las violaciones halladas (resultado inicial):**

El resultado inicial luego de verificar el código con PMD es de **661** lo que indica una gran cantidad de violaciones:  


Estas 661 se reparten en:

* Warnign violations  : 0
* Important violations  : 18
* Urgent violations  : 596
* Critical violations  : 45
* Blocker violations : 2

**Urgent violations** 

a mayor cantidad de violaciones son del grupo verde el cual corresponde en mayor parte a violaciones que afectan la estructura, rendimiento, y mantenibilidad del código. Estas violaciones no suelen provocar fallos críticos, pero sí impactan negativamente en la claridad y eficiencia del código.

Las que más se repiten son:

**MethodArgumentCouldBeFinal** (111):



Esto indica que un argumento del método puede ser declarado **final** porque no se reasigna ni modifica en el cuerpo del método.

**LocalVariableCouldBeFinal** (106):



Esto indica que hay alguna variable local dentro de un método que puede ser declarada **final** porque no se modifica después de su inicialización.

**CommentedRequired** (115):



Esto indica que falta comentarios ya sea en un método, instancia, atributo, clase o método.

**Critical violations** 

El segundo grupo con más violaciones es el azul que corresponde a violaciones referentes a errores graves como problemas de concurrencia, estructuras de datos no sincronizadas en contextos multihilo, y otros fallos que pueden afectar la estabilidad del sistema o producir bloqueos. Esto hace que sea importante resolver estos problemas para garantizar el correcto funcionamiento de la aplicación.

Para este grupo solo hay un tipo de violación:  
**SystemPrintln** (45):

  
La violación SystemPrintln en PMD señala que el uso de System.out.print o System.err.print en el código suele ser indicativo de código de depuración que quedó de forma accidental, lo cual es una práctica no recomendada, especialmente en aplicaciones de producción.

Por motivos de simplicidad hemos decidido ignorar este tipo de violación ya que consideramos que para el trabajo que estamos haciendo no hace falta cambiar los prints por loggers, la única parte que si manejara este tipo de salida serán las pruebas de aceptación.

**Important violations** 

El tercer grupo correspondiente a las rosadas son violaciones menos críticas que afectan el estilo o la legibilidad del código. Incluyen recomendaciones de buenas prácticas, como nombres de variables descriptivos y organización de código.

Los tipos de violaciones que nos salen son:

**UseVarargs** (14):



Esta violación sugiere que en métodos o constructores donde se acepta un número variable de argumentos (generalmente representado como un arreglo), se debería considerar usar la sintaxis varargs (...) en lugar de requerir que el usuario pase explícitamente un arreglo. Consideramos que usar varargs es una opción viable y hace la experiencia más amigable al usuario, pero por cuestiones de tiempo no lo implementaremos en el proyecto actual.

**ShortClassName** **y UselessParentheses**(2):



La primera indica que no es recomedable tener clases con un nombre con longitud menor a 5 caracteres como glue o tilt, ambos consideramos que no es tan importante y que el nombre actual es bastante acorde con lo que hace la clase. La segunda indica que hay paréntesis que se pueden quitar sin que se afecte la lógica del código.

**Blocker violations**

El cuarto y último grupo de violaciones que tenemos son las rojas, estas son las más prioritarias ya que pueden afectar gravemente la funcionalidad o seguridad del código y pueden llevar a errores de ejecución. Ejemplos incluyen errores de sincronización, NullPointerException, o condiciones que afectan directamente la ejecución.

Los tipos de violaciones que no salen son:

**ClassWithOnlyPrivateConstructorsShouldBeFinal** (1):  


Esta violación indica que hay clases que tienen solo constructores privados, lo que significa que no pueden ser extendidas desde fuera de su unidad de compilación (el archivo en el que se encuentran). Dado que no se puede crear una subclase que invoque el constructor de la clase original, esta se considera candidata para ser declarada final.

**FieldNamingConventions** (1):

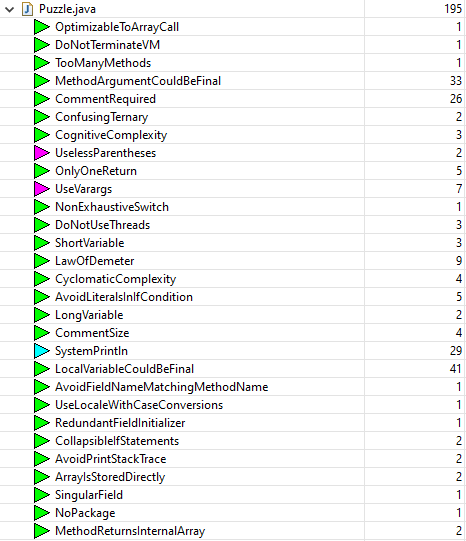


Esta violación se refiere a las convenciones de nomenclatura para las declaraciones de campos (variables) en el código. Esta regla verifica si los nombres de las variables cumplen con las expresiones regulares (regex) configuradas que se aplican a su tipo específico, como constantes (static final), campos estáticos, y campos finales.

**Violaciones encontradas por clase (resultado inicial):**

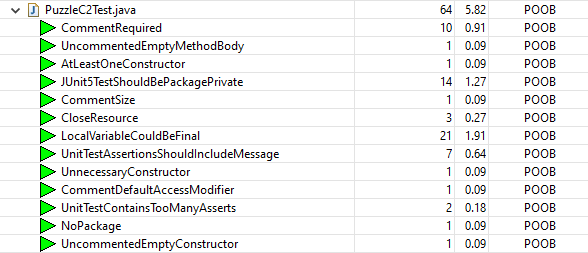
1. **Puzzle:** Para esta clase tenemos un total de 195 violaciones que se reparten en:

* Warnign violations  : 0
* Important violations  : 9
* Urgent violations  : 157
* Critical violations  : 29
* Blocker violations : 0

Entre las cuales tenemos 29 tipos distintos:  


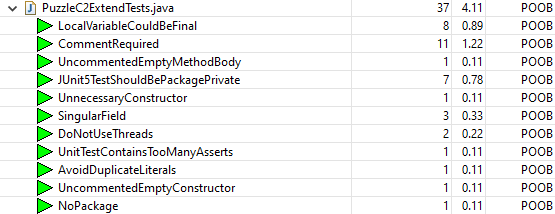
1. **PuzzleC2Test:** Para esta clase tenemos un total de 64 violaciones que se reparten en:

* Warnign violations  : 0
* Important violations  : 0
* Urgent violations  : 64
* Critical violations  : 0
* Blocker violations : 0



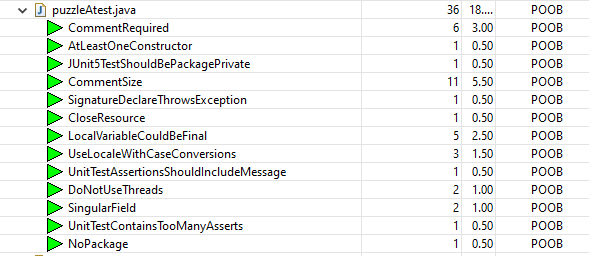
1. **PuzzleC2ExtendTests:** Para esta clase tenemos un total de 37 violaciones repartidas en:

* Warnign violations  : 0
* Important violations  : 0
* Urgent violations  : 37
* Critical violations  : 0
* Blocker violations : 0



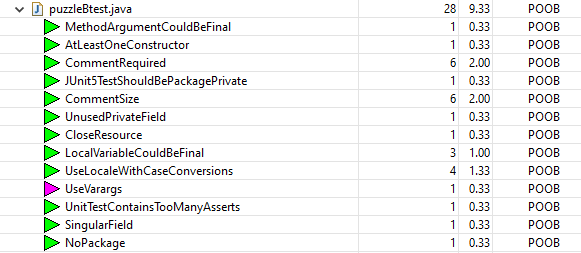
1. **PuzzleAtest:** Para esta clase tenemos un total de 36 violaciones repartidas en:

* Warnign violations  : 0
* Important violations  : 0
* Urgent violations  : 36
* Critical violations  : 0
* Blocker violations : 0



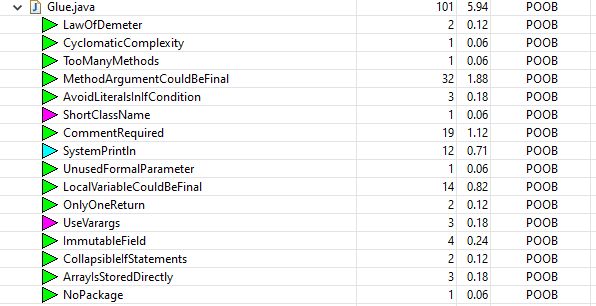
1. **PuzzleBTest**: Para esta clase tenemos un total de 28 violaciones repartidas en:

* Warnign violations  : 0
* Important violations  : 1
* Urgent violations  : 27
* Critical violations  : 0
* Blocker violations : 0



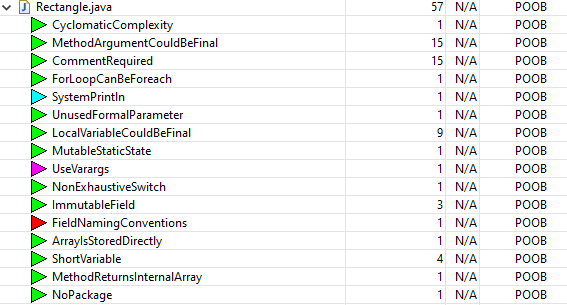
1. **Glue:** Para esta clase tenemos un total de 101 excepciones repartidas en:

* Warnign violations  : 0
* Important violations  : 4
* Urgent violations  : 85
* Critical violations  : 12
* Blocker violations : 0



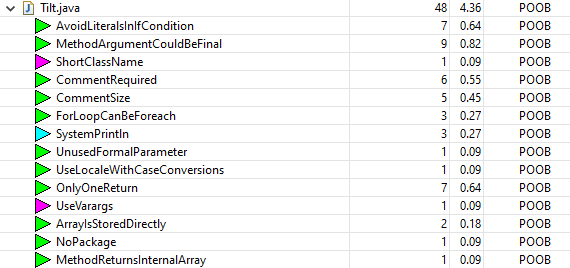
1. **Rectangle:** En esta clase hay un total de 56 violaciones repartidas así:

* Warnign violations  : 0
* Important violations  : 1
* Urgent violations  : 53
* Critical violations  : 1
* Blocker violations : 1



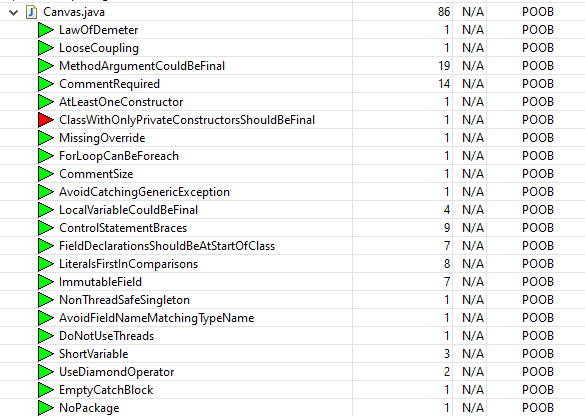
1. **Tilt:** Esta clase tiene 48 violaciones en total repartidas así:

* Warnign violations  : 0
* Important violations  : 2
* Urgent violations  : 43
* Critical violations  : 3
* Blocker violations : 0



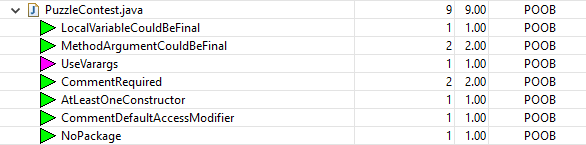
1. **Canvas:** Esta clase cuenta con 85 excepciones repartidas de la siguiente forma:

* Warnign violations  : 0
* Important violations  : 0
* Urgent violations  : 84
* Critical violations  : 0
* Blocker violations : 1



1. **PuzzleContest:** Esta clase tiene 9 excepciones en total repartidas así:

* Warnign violations  : 0
* Important violations  : 1
* Urgent violations  : 8
* Critical violations  : 0
* Blocker violations : 0



**Resumen general de las violaciones halladas (resultado final):**

Luego de observar la gran cantidad de violaciones que había en nuestro código y el corto tiempo que nos quedaba decidimos seguir el orden de prioridad que indica PMD y darle solución al 100% de las violaciones color rojo y azul que son las más prioritarias y con el tiempo restante intentar solucionar la mayor cantidad de violaciones verdes que alcanzáramos, el resultado final es el siguiente.

El resultado final luego de verificar el código con PMD y manejar las violaciones dichas es de **279** lo que indica que reducimos en más de la mitad el número total de las que había inicialmente.



Estas 279 se reparten de la siguiente forma:

* Warnign violations  : 0
* Important violations  : 18
* Urgent violations  : 261
* Critical violations  : 0
* Blocker violations : 0

Como se puede ver ya no tenemos ni Critical o Blocker violations lo cual asegura que como mínimo el programa corra con normalidad sin miedo a que salte algún error, el mayor número de excepciones siguen siendo las Urgents pero consideramos que la mayoría de las excepciones de este tipo son más documentación en parte no necesaria para las características de nuestro proyecto y algo similar ocurre con las Importants pero en menor medida.

A continuación, mostraremos como manejamos cada uno de los tipos de violaciones para llegar a reducir el número total poniendo un ejemplo por cada una:

**Revisión de violaciones rojas (2):**

**ClassWithOnlyPrivateConstructorsShouldBeFinal** (1):  


Para no tener más esta violación tuvimos que hacer un cambio en la siguiente línea de código de la clase **Canvas:**



Lo que hicimos es ponerle el atributo final, esto se debe a que todos los constructores de la clase **Canvas** son privados esto implica que no se puede crear una subclase que llame al constructor de la clase **Canvas**, lo que la convierte en una clase que no necesita ser extendida.



**FieldNamingConventions** (1):



Para no tener más esta violación tuvimos que hacer un cambio en la siguiente línea de código de la clase **Rectangle:**  


Lo que hicimos es ponerle final luego del static ya que es un valor que nunca va a cambiar porque siempre un objeto de tipo rectángulo va a tener cuatro bordes.



**Revisión de violaciones azules:**

Como dijimos arriba decidimos no mover los Systems.out.print que teníamos y que eran los que nos estaban dando error en las cuatro clases donde nos salían **(Glue, Puzzle, Rectangle y Tilt)** y solamente cambiar esto en las pruebas de aceptación que creamos. Para ello tuvimos que manejar los mensajes de la siguiente forma:

**PuzzleAtest:**

Importamos logger y lo definimos como un atributo de la clase.

**import** java.util.logging.Logger;



Luego cada vez que queremos mostrarle algo al usuario en consola no usamos prints sino loggers:









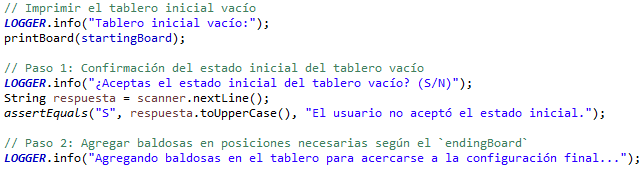
De esta forma nos aseguramos de que este tipo de violación no salga en la clase.

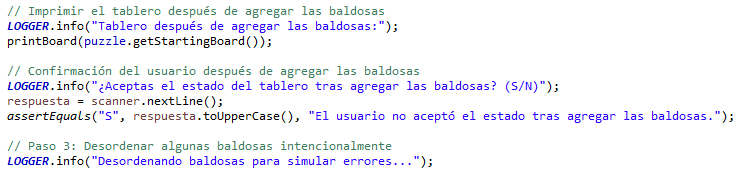
**PuzzleBtest:**

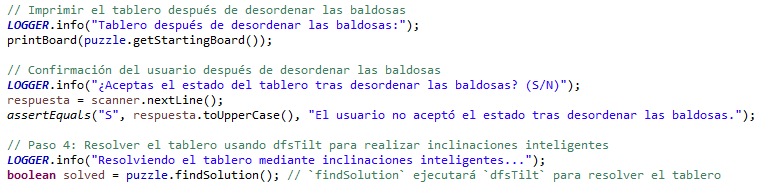
Similar a **PuzzleAtest** importamos a logger y lo definimos como un atributo de la clase.





Y cada vez que queramos decirle algo al usuario lo hacemos por medio de un logger:  






De esta forma no saldrá este tipo de violación en esta clase.

Por últimos le decimos deshabilitar esta violación para que no se mostrara en las otras clases ya sabiendo que las clases referentes a las pruebas no la tienen.

**Revisión de algunas violaciones verdes:**

Para empezar a manejar estas violaciones y dado que eran muchas decidimos empezar por las que se repetían más:  
  
**MethodArgumentCouldBeFinal** (111):



Para quitar este tipo de violación añadimos final a algunas partes en las que el argumento no se modificaba a lo largo del método lo que contribuye a la inmutabilidad y ayuda a prevenir errores accidentales en la lógica.

Ejemplo:





**LocalVariableCouldBeFinal** (106):



Para quitar esta violación repetimos un proceso similar pero ahora con variables por fuera de los métodos:

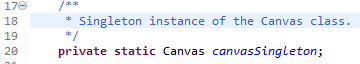




**CommentedRequired** (115):



Por último, para quitar esta violación debemos comentar arriba de la línea indicando que es lo que hace cada cosa.  

Luego de estar un rato quitando una por una cada violación del mismo estilo que se presentó arriba llegamos a la conclusión de que este tipo de violación no tiene la suficiente importancia como para seguir gastando tiempo en corregirlas, es por esto por lo que al final decidimos ignorar desde el entorno de eclipse este tipo de violaciones.

Para concluir, el uso de la extensión PMD resultó de gran utilidad, ya que permitió realizar ajustes en el código que podrían prevenir posibles fallos graves en el proyecto. Además, ofreció información valiosa que será de gran ayuda en proyectos futuros, como la importancia de mantener convenciones en la nomenclatura. Aunque puede parecer tedioso, el nombrado claro y consistente es esencial para mejorar la legibilidad y mantener la claridad en el código, facilitando así el trabajo en equipo y el mantenimiento a largo plazo.