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:

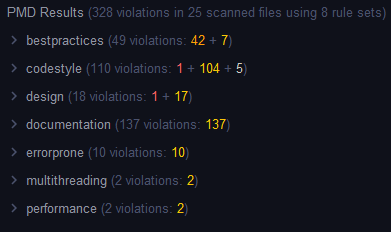
Juan Sebastián Frasica Galeano

María Irma Diaz Rozo

El propósito de este informe es documentar los resultados del análisis estático realizado sobre el código fuente del proyecto final basado en el juego de “Plants vs Zombies” utilizando la extensión PMD en el entorno de IntelliJ. 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 se parten en dos grandes directorios (dominio y presentación). Por el lado de dominio se escanearon un total de 25 clases en búsqueda de violaciones de los siguientes tipos:

* BestPractices
* CodeStyle
* Design
* Documentation
* ErrorProne
* Multithreading
* Performance

Luego de escanear arrojo los siguientes resultados:  


En total se encontraron 328 violaciones entre las cuales se resaltan las siguientes por ser las más críticas según el mismo PMD.

**BestPractices:**

**SystemPrintln:** Las referencias a System.(out|err).print generalmente están destinadas a fines de depuración y pueden permanecer en el código base incluso en el código de producción. Al usar un registrador, se puede habilitar/deshabilitar este comportamiento a voluntad (y por prioridad) y evitar obstruir el sistema.

Este tipo de violación se repitió 42 veces en distintas clases de dominio y en algunas fue de gran utilidad como por ejemplo esta:  


La cual nos hizo acordar del siguiente método que usamos para depurar y se nos olvidó quitar:  


**CodeStyle:**

**ClassNamingConventions:** Convenciones de nomenclatura configurables para declaraciones de tipos. Esta regla informa declaraciones de tipos que no coinciden con la expresión regular que se aplica a su tipo específico (por ejemplo, enumeración o interfaz). Cada expresión regular se puede configurar a través de propiedades.

Este tipo violación se encontró solo una vez en la clase brainManager:  

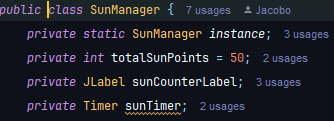

Y se solucionó poniendo la primera letra en mayúscula quedando como BrainManager:



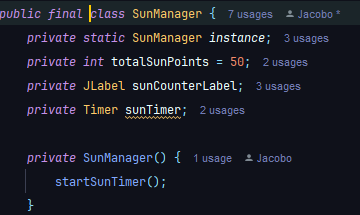
**Design:**

**ClassWithOnlyPrivateConstructorsShouldBeFinal:** Esta clase sólo tiene constructores privados y puede ser final. Esta violacion informa las clases que pueden convertirse en finales porque de todos modos no se pueden extender desde fuera de su unidad de compilación. Esto se debe a que todos sus constructores son privados, por lo que una subclase no podría llamar al super constructor.

El error se encontraba en la clase SunManager la cual al tener solo constructores privados podía ser final, y se solucionó la violación aplicando esto:  
**Antes:**



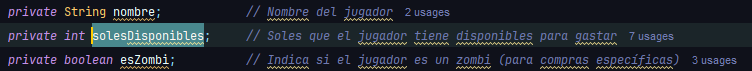
**Despues:**



**Documentation:**

En el apartado de documentación la mayor parte de las violaciones fueron por:

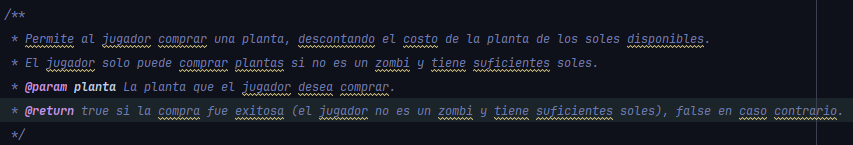
**CommentRequired:** Indica si los comentarios son requeridos o innecesarios en determinadas partes del código.

Esta violacion se repitió un total de 98 veces, pero la mayor parte eran en zonas donde se definían atributos y que tenían su documentación en forma de comentario como, por ejemplo:  


En la que si bien es cierto que no hay comentarios formales (/\*\*) se entiende cada parámetro y lo que hace en base a sus comentarios (//)

**CommentSize:** Indica que la dimensión de una línea es muy larga.

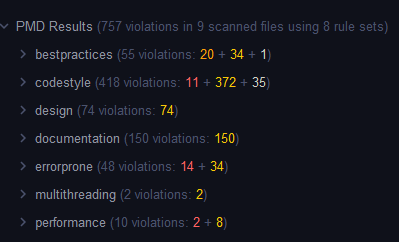
Esta violación nos salió 39 veces indicando que una línea era demasiado extensa, pero consideramos que la mayoría eran por 3 o 4 caracteres que se salían del límite y que se podían ver perfectamente en una pantalla convencional sin necesidad de mover de forma horizontal la pantalla.



Para los otros 3 tipos de violaciones (ErrorProne, Multithreading y performace) eran violaciones menores que no valen la pena mencionar.

Por el lado de presentación se analizaron un total de 9 clases y se encontraron 757 violaciones de los siguientes tipos:

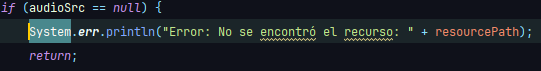
* BestPractices
* CodeStyle
* Design
* Documentation
* ErrorProne
* Multithreading
* Performance

El escaneo arrojo los siguientes resultados:  


**BestPractices:**

**SystemPrintln:** Las referencias a System.(out|err).print generalmente están destinadas a fines de depuración y pueden permanecer en el código base incluso en el código de producción. Al usar un registrador, se puede habilitar/deshabilitar este comportamiento a voluntad (y por prioridad) y evitar obstruir el sistema.

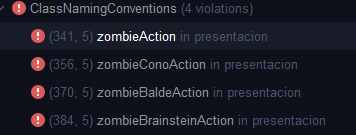
Este tipo de violación se repitió un total de 20 veces, pero a diferencia de dominio estas eran para el manejo de errores y dado nuestra falta de una clase que manejara excepciones era la única forma que teníamos de mostrarlos en pantalla, debido a esto no podíamos quitarlos del código.



**CodeStyle:**

**ClassNamingConventions:** Convenciones de nomenclatura configurables para declaraciones de tipos. Esta regla informa declaraciones de tipos que no coinciden con la expresión regular que se aplica a su tipo específico (por ejemplo, enumeración o interfaz). Cada expresión regular se puede configurar a través de propiedades.

Este tipo de violacion se repitió 4 veces dentro de la clase tableroPvsP y similar a dominio s soluciono poniendo la primera letra en mayúscula en los 4 casos.







**FieldNamingConventions:** Convenciones de nombres configurables para declaraciones de campos. Esta regla informa las declaraciones de variables que no coinciden con la expresión regular que se aplica a su tipo específico (por ejemplo, constantes (final estática), constante de enumeración, campo final). Cada expresión regular se puede configurar a través de propiedades.

Esta violación salto una vez en un Jlabel que estaba como “Menú” el cual no se estaba ni siquiera utilizando así que simplemente se eliminó:  


**LocalVariableNamingConventions:** Convenciones de nombres configurables para declaraciones de variables locales y otras variables de ámbito local. Esta regla informa las declaraciones de variables que no coinciden con la expresión regular que se aplica a su tipo específico (por ejemplo, variable final o parámetro de cláusula catch). Cada expresión regular se puede configurar a través de propiedades.

Esta violación salto un total de 6 veces todas porque el nombre de la variable comenzaba con mayúscula, para arreglarlo le pusimos la primera letra en minúscula:  












**ErrorProne:**

**ConstructorCallsOverridableMethod:** Informa de las llamadas a métodos reemplazables en `this` durante la inicialización del objeto. Se invocan en un objeto construido de forma incompleta y pueden ser difíciles de depurar si se reemplazan.

Esto se debe a que la subclase suele suponer que la superclase está completamente inicializada en todos los métodos. Si ese no es el caso, pueden aparecer errores en el constructor; por ejemplo, algunos campos que aún son nulos pueden provocar una NullPointerException o almacenarse en otro lugar para que exploten más tarde.

Para evitar este problema, utilice únicamente métodos que sean estáticos, privados o finales en los constructores.

Este tipo de violación salio un total de 14 veces la mayor de las veces en métodos como prepareElements en los cuales creábamos las diferentes cosas que usaríamos a nivel visual.







Por cuestiones de tiempo y dado que la violacion implicaba cambiar cosas muy importantes a nivel visual decidimos no tocar nada y dejarlas así, obviamente teniendo en cuenta estas consideraciones para futuros proyectos que realicemos. Mas allá de estas violaciones no había otras que de similar importancia que fueran necesarias de revisar.

**Performance:**

**AvoidFileStream:** Evite crear instancias de FileInputStream, FileOutputStream, FileReader o FileWriter

Las clases FileInputStream y FileOutputStream contienen un método finalizador que provocará pausas en la recolección de elementos no utilizados.

Los constructores FileReader y FileWriter crean instancias de FileInputStream y FileOutputStream, lo que nuevamente provoca problemas de recolección de elementos no utilizados mientras se invocan los métodos finalizadores.

\* Utilice `Files.newInputStream(Paths.get(fileName)) ` en lugar de `new FileInputStream(fileName)`.

\* Utilice `Files.newOutputStream(Paths.get(fileName)) ` en lugar de `new FileOutputStream(fileName)`.

\* Utilice `Files.newBufferedReader(Paths.get(fileName)) ` en lugar de `new FileReader(fileName)`.

\* Utilice `Files.newBufferedWriter(Paths.get(fileName)) ` en lugar de `new FileWriter(fileName)`.

Este error nos salio un total de 2 veces en los métodos de guardar/cargar partida para la parte de persistencia:  


Siguiendo las recomendaciones intentamos cambiar esa parte por lo que nos decía la violación pero nos tocaba cambiar algunas cosas en otras clases así que por cuestiones de tiempo lo dejamos igual, seguiremos las indicaciones para futuros proyectos que realicemos.

Para los otros 3 tipos de violaciones (Design , Documentation y Multithreading) eran violaciones menores que no valen la pena mencionar.

**En conclusión:**

El análisis estático realizado utilizando **PMD** permitió identificar y corregir una serie de problemas en el código fuente, lo que contribuyó significativamente a mejorar su calidad, mantenibilidad y desempeño. Entre los principales hallazgos se destacaron:

1. **Problemas de Código Duplicado**: Se detectaron fragmentos de código redundantes que, al ser refactorizados, redujeron la complejidad del proyecto y facilitaron la centralización de la lógica común.
2. **Buenas Prácticas de Programación**: PMD identificó violaciones de convenciones de nomenclatura, falta de documentación, y uso incorrecto de estructuras como bucles y condicionales, lo que permitió alinearnos con los estándares de codificación recomendados.
3. **Problemas de Desempeño y Complejidad**: Se destacaron métodos y clases con una complejidad ciclomática elevada, los cuales fueron simplificados para mejorar la legibilidad y facilitar el mantenimiento.
4. **Errores Potenciales**: El análisis ayudó a detectar posibles **NullPinterExceptions**, fugas de recursos y otras fallas comunes, lo que incrementó la robustez del sistema.

Como resultado del proceso, el código ahora presenta una estructura más limpia y eficiente, alineada con las mejores prácticas del desarrollo en Java. Este análisis también permitió mejorar el entendimiento general del sistema y establecer una base sólida para su evolución futura.

Recomendamos incorporar **PMD** como parte del flujo de integración continua para mantener la calidad del código a lo largo del ciclo de vida del proyecto y prevenir la introducción de nuevas anomalías.