

UT4. Optimización y Documentación

ENTORNOS DE DESARROLLO

MARCOS FERNÁNDEZ SELLERS



Introducción

❖ Durante todo el proceso de desarrollo software hay que realizar optimizaciones y documentaciones.

❖ ¿Refactorización?

❖ ¿Optimización?

Refactorización

- ❖ La refactorización es una disciplina técnica, que consiste en realizar pequeñas transformaciones en el código de un programa, para mejorar la estructura **sin que cambie el comportamiento ni funcionalidad del mismo**.
- ❖ Su objetivo es mejorar la estructura interna del código.
- ❖ Es una tarea que pretende limpiar el código minimizando la posibilidad de introducir errores.

Refactorización

Otra definición:

- ❖ La refactorización es una técnica de ingeniería del software para organizar el código de una forma más eficiente y fácil de entender , de tal forma que si estamos trabajando con más programadores y necesitan ver nuestro código no les resulte complicado saber qué hace y pueda ser reutilizado sin complicaciones.
- ❖ En pocas palabras, la refactorización “limpia el código”.

Refactorización

- ❖ Mejora el diseño software.
- ❖ Software más fácil de entender.
- ❖ Mantenimiento más sencillo.
- ❖ Ayuda a encontrar errores.
- ❖ Aumenta la velocidad del programa.

Refactorización

- ❖ Cuando se refactoriza se está mejorando el diseño del código después de haberlo escrito.
- ❖ Podemos partir de un mal diseño y, aplicando la refactorización, llegaremos a un código bien diseñado.
- ❖ Cada paso es simple, por ejemplo mover una propiedad desde una clase a otra, convertir determinado código en un nuevo método, etc.
- ❖ La acumulación de todos estos pequeños cambios pueden mejorar de forma ostensible el diseño.

Refactorización - Concepto

- ❖ El concepto de refactorización de código, se base en el concepto matemático de factorización de polinomios.

FACTORIZACIÓN DE POLINOMIOS

$$X^2 - 1 = (X + 1)(X - 1)$$

- ❖ Refactorización: Cambio hecho en la estructura interna del software para hacerlo más fácil de entender y fácil de modificar **sin modificar su comportamiento**.

Refactorización - Concepto

Ejemplos:

- ❖ Extraer método: Convertir un fragmento de código que se repite varias veces en un método nuevo independiente.
- ❖ Campos encapsulados: Se aconseja crear métodos getter y setter.

Refactorización - Concepto

- ❖ Hay que diferenciar la refactorización de la optimización.
- ❖ En ambos procesos, se pretende mejorar la estructura interna de una aplicación o componente, sin modificar su comportamiento.
- ❖ Sin embargo, cuando se optimiza, se persigue una mejora del rendimiento, por ejemplo, mejorar la velocidad de ejecución o consumir menos memoria, pero esto puede hacer un código más difícil de entender.
- ❖ No obstante, según como se aplique la refactorización también se puede estar optimizando de manera implícita.

Refactorización - Concepto

❖ Hay que resaltar que la refactorización no cambia el comportamiento observable del software. El software sigue cumpliendo la misma función que hacía antes. Ningún usuario, ya sea usuario final u otro programador, podrá determinar qué cosas han cambiado.



Cuándo refactorizar

- ❖ La refactorización se debe ir haciendo mientras se va realizando el desarrollo de la aplicación.
- ❖ Piattini y García (2003) analizan los síntomas que indican la necesidad de refactorizar , a los que Martin Fowler with Kent Beck, Refactoring , 1999) llamaron bad smells (malos olores)

Malos olores (Bad Smells)

❖ Código duplicado

Es la principal razón para refactorizar. Si se detecta el mismo código en más de un lugar, se debe buscar la forma de extraerlo y unificarlo.

❖ Métodos muy largos

Cuanto más largo es un método más difícil es de entender. Un método muy largo normalmente está realizando tareas que deberían ser responsabilidad de otros. El método se debe identificar y descomponer en otros más pequeños.

En POO cuanto más corto es un método más fácil es reutilizarlo.

Malos olores (Bad Smells)

❖ Clases muy grandes

Si una clase intenta resolver muchos problemas, tendremos una clase con demasiados métodos, atributos o incluso instancias. La clase está asumiendo demasiadas responsabilidades. Hay que intentar hacer clases más pequeñas, de forma que cada una trate con un **conjunto pequeño de responsabilidades bien delimitadas**.

Malos olores (Bad Smells)

❖ Lista de parámetros extensa

En POO no se suelen pasar muchos parámetros a los métodos, sólo los necesarios para que el objeto involucrado consiga lo que necesita. Tener demasiados parámetros puede estar indicando un problema de encapsulación de datos o la necesidad de crear una clase de objetos a partir de varios de esos parámetros, y pasar ese objeto como argumento en vez de todos los parámetros.

Malos olores (Bad Smells)

❖ Envidia de funcionalidad

Se observa este síntoma cuando tenemos un método que utiliza más cantidad de elementos de otra clase que de la suya propia. Se suele resolver el problema pasando el método a la clase cuyos elementos utiliza más.

Refactorización - Limitaciones

- ❖ Están claras las ventajas de refactorización.
- ❖ Pero, ¿cuáles serán las limitaciones?

Refactorización - Limitaciones

- ❖ Un área problemática de la refactorización son las bases de datos.
- ❖ Una base de datos presenta muchas dificultades para poder ser modificada, dado la gran cantidad de interdependencias que soporta.
- ❖ Cualquier modificación que se requiera de las bases de datos, incluyendo modificación de esquema y migración de datos, puede ser una tarea muy costosa.
- ❖ Es por ello que la refactorización de una aplicación asociada a una base de datos, siempre será limitada, ya que la aplicación dependerá del diseño de la base de datos.

Refactorización - Limitaciones

- ❖ Otra limitación es cuando cambiamos interfaces.
- ❖ Cuando refactorizamos, estamos modificando la estructura interna de un programa o de un método.
- ❖ El cambio interno no afecta al comportamiento ni a la interfaz. Sin embargo, si renombramos un método, hay que cambiar todas las referencias que se hacen a él.
- ❖ Siempre que se hace esto se genera un problema si es una interfaz pública.
- ❖ Una solución es mantener las dos interfaces, la nueva y la vieja, ya que si es utilizada por otra clase o parte del proyecto, no podrá referenciarla.

Refactorización – Patrones de refactorización

A los métodos de refactorización también se les llama patrones de refactorización o catálogos de refactorización. Podemos refactorizar:

- ❖ una clase
- ❖ una variable
- ❖ una expresión
- ❖ un bloque de instrucciones
- ❖ un método
- ❖ etc

Refactorización – Patrones de refactorización

Para refactorizar:

1. Se selecciona el elemento,
2. Se pulsa el botón derecho del ratón,
3. Se selecciona Refactor o Reestructurar,
4. Seguidamente se selecciona el método de refactorización.

A continuación, se muestran algunos de los métodos más comunes:

Refactorización – Patrones de refactorización

- ❖ **Renombrado** (rename): Este patrón nos indica que debemos cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.
- ❖ **Sustituir bloques de código por un método**: Este patrón nos aconseja sustituir un bloque de código, por un método. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método.

Refactorización – Patrones de refactorización

- ❖ **Campos encapsulados:** Se aconseja crear métodos getter y setter, (de asignación y de consulta) para cada campo que se defina en una clase. Cuando sea necesario acceder o modificar el valor de un campo, basta con invocar al método getter o setter según convenga.
- ❖ **Mover la clase:** Si es necesario, se puede mover una clase de un paquete a otro, o de un proyecto a otro. La idea es no duplicar código que ya se haya generado. Esto impone la actualización en todo el código fuente de las referencias a la clase en su nueva localización.

Refactorización – Patrones de refactorización

- ❖ **Borrado seguro:** Se debe comprobar, que cuándo un elemento del código ya no es necesario, se han borrado todas las referencias a él que había en cualquier parte del proyecto.
- ❖ **Cambiar los parámetros del proyecto:** Nos permite añadir nuevos parámetros a un método y cambiar los modificadores de acceso.
- ❖ **Extraer la interfaz:** Crea una nueva interfaz de los métodos public non-static seleccionados en una clase o interfaz.
- ❖ **Mover del interior a otro nivel:** Consiste en mover una clase interna a un nivel superior en la jerarquía.

Refactorización – Analizadores de código

- ❖ Cada IDE incluye herramientas de refactorización y analizadores de código. En el caso de software libre, existen analizadores de código que se pueden añadir como complementos a los entornos de desarrollo. Respecto a la refactorización, los IDE ofrecen asistentes que de forma automática y sencilla, ayudan a refactorizar el código.

Refactorización – Analizadores de código

- ❖ El **análisis estático de código** es un proceso que tiene como objetivo evaluar el software, sin llegar a ejecutarlo.
- ❖ Esta técnica se va a aplicar directamente sobre el código fuente, para poder obtener información que nos permita mejorar la base de código, pero sin que se modifique la semántica.
- ❖ Los analizadores de código, son las herramientas encargadas de realizar esta labor. El analizador estático de código recibirá el código fuente de nuestro programa, lo procesará intentando averiguar la funcionalidad del mismo, y nos dará sugerencias, o nos mostrará posibles mejoras.

Refactorización – Analizadores de código

- ❖ Los analizadores de código incluyen analizadores léxicos y sintácticos que procesan el código fuente y de un conjunto de reglas que se deben aplicar sobre determinadas estructuras. Si el analizador considera que nuestro código fuente tiene una estructura mejorable, nos lo indicará y también nos comunicará la mejora a realizar.
- ❖ Las principales funciones de los analizadores es encontrar partes del código que puedan reducir el rendimiento, provocar errores en el software, tener una excesiva complejidad, complicar el flujo de datos, crear problemas de seguridad.

Refactorización – Analizadores de código

- ❖ El análisis puede ser automático o manual. El automático, los va a realizar un programa, que puede formar parte de la funcionalidad de un entorno de desarrollo, por ejemplo el SonarLint en NetBeans, o manual, cuando es una persona.
- ❖ El análisis automático reduce la complejidad para detectar problemas de base en el código, ya que los busca siguiendo una serie de reglas predefinidas. El análisis manual, se centra en apartados de nuestra propia aplicación, como comprobar que la arquitectura de nuestro software es correcta.

Refactorización – Analizadores de código

❖ Analizadores Web:

SCRUTINIZER



- PHP, Python y Ruby soportados
- 14 días de prueba gratis
- Precio/Mes:
 - Plan Basic: 49 €
 - Plan Professional: 99 €
 - Plan Unlimited: 199 €

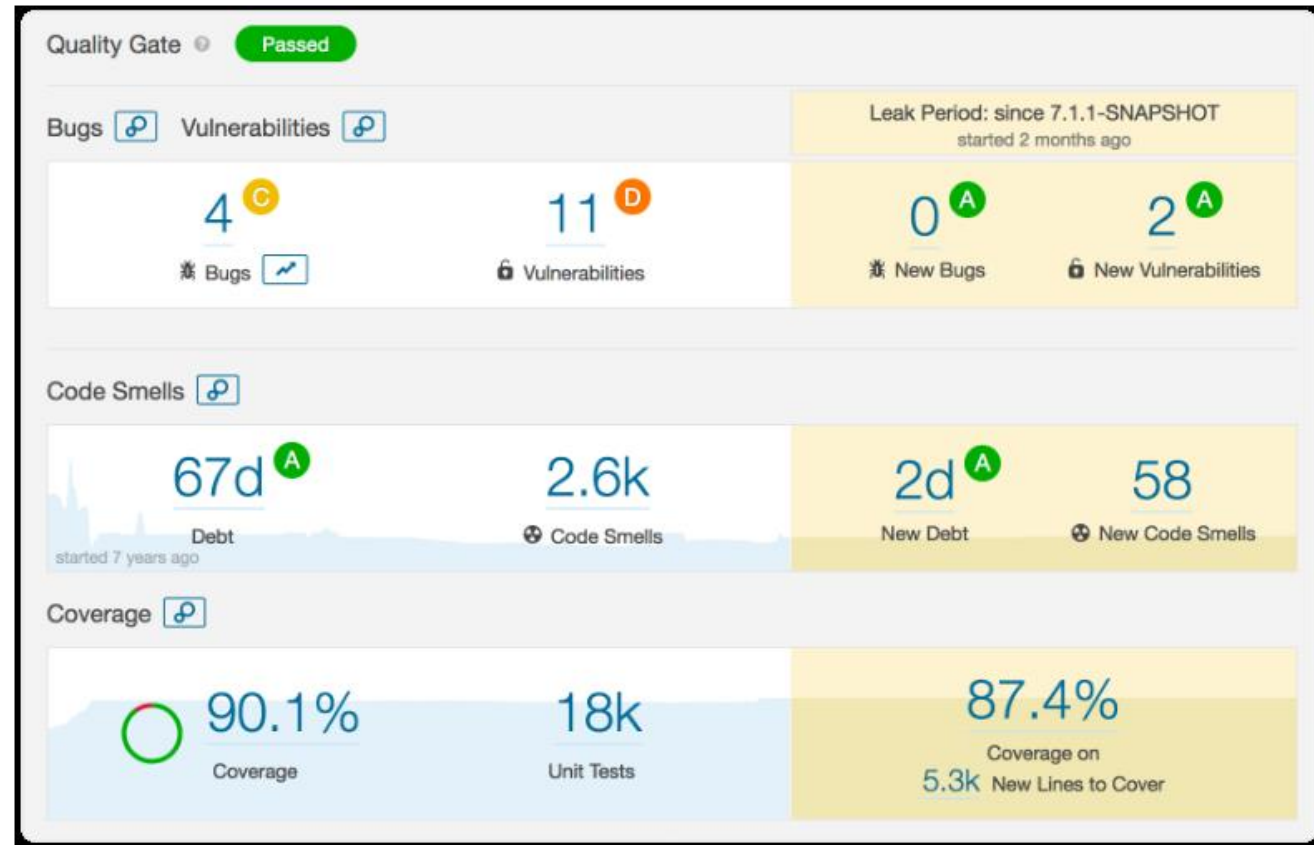
SONARQUBE



- Más de 20 lenguajes soportados
- Precio/Mes:
 - Plan Open Source: 0 \$
 - Plan Private Projects: Desde 5 \$

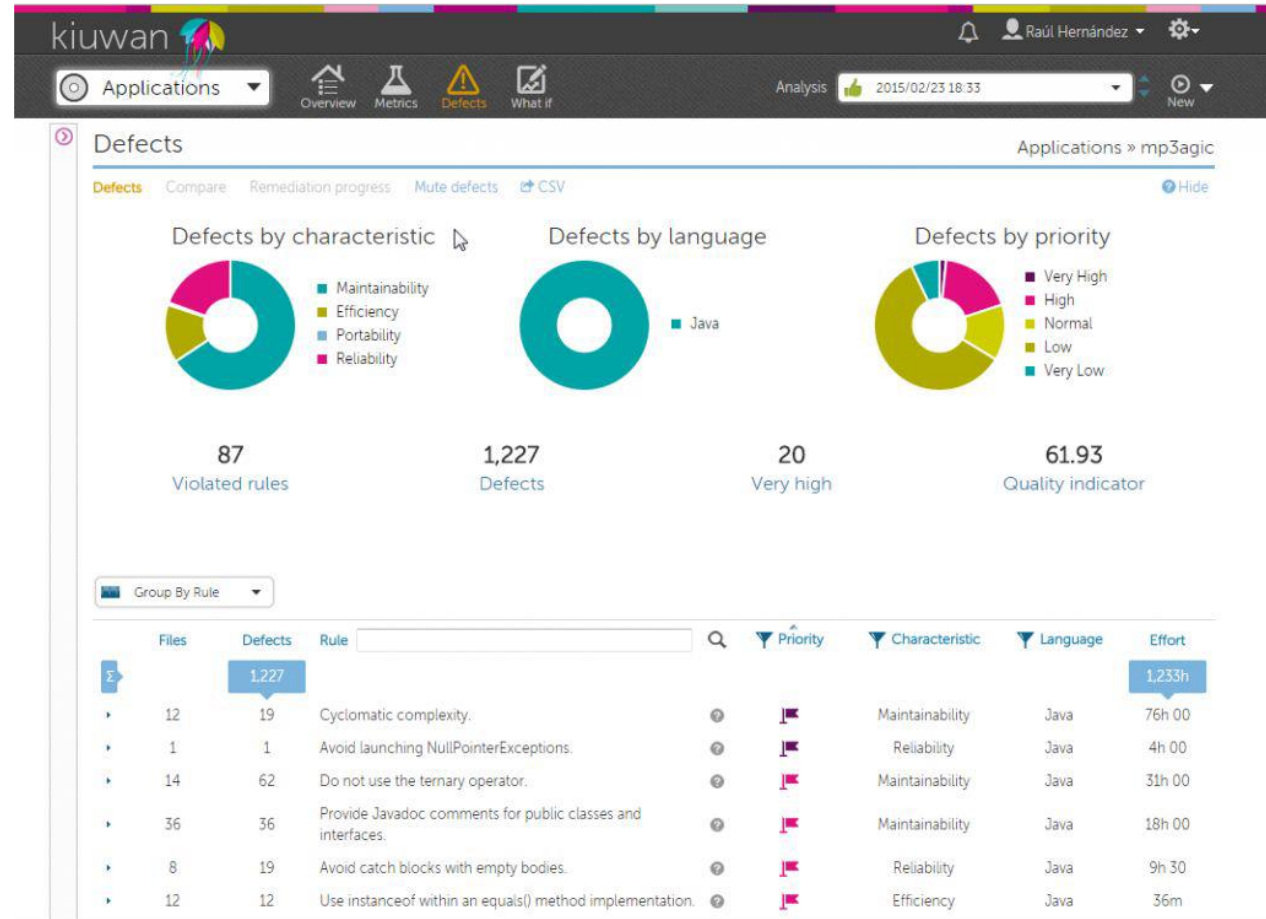
Refactorización – Analizadores de código

❖ SonarQube también disponible para uso local bajo licencia open source.



Refactorización – Analizadores de código

❖ Otra alternativa (en este caso de pago), sería Kiuwan.



Refactorización – Analizadores de código

❖ Tomando como base el lenguaje de programación Java, nos encontramos disponibles un conjunto de analizadores para integrar con nuestros entornos de desarrollo, entre otros:

❖ **PMD**

❖ **CHECKSTYLE**

❖ **SONARLINT**

PMD – Analizadores de código

Es una herramienta que audita el código Java encontrando problemas como:

- ❖ Código muerto: variables y métodos (sólo privados) no usados
- ❖ Código no óptimo: uso de Strings,...
- ❖ Expresiones complejas: uso de if innecesarios, for,...
- ❖ Código duplicado
- ❖ Posibles bugs
- ❖ PMD además permite definir nuevas reglas.
- ❖ Referencia: <https://pmd.github.io>



Checkstyle – Analizadores de código

- ❖ Checkstyle es una herramienta de análisis de código estático que se utiliza en el desarrollo de software para comprobar si el código fuente de Java cumple con las reglas de codificación.
- ❖ Desarrollado originalmente por Oliver Burn en 2001, el proyecto es mantenido por un equipo de desarrolladores de todo el mundo.
- ❖ Referencia: <https://checkstyle.sourceforge.io>



SonarLint – Analizadores de código

❖ SonarLint es una extensión para diferentes IDE de desarrollo que permite encontrar incidencias en el código de forma automática mientras se codifica, mostrando las diferentes incidencias para poder localizarlas fácilmente, así como una explicación de la misma y su posible solución.

❖ Referencia: <https://www.sonarlint.org>



SonarLint – Analizadores de código

```
mxCellState gatewayState = graph.getView().getState(gatewayVertex);

mxPoint northPoint = new mxPoint(gatewayState.getX() + (gatewayState.getWidth() / 2, gatewayState.getY());
mxPoint southPoint = new mxPoint(gatewayState.getX() + (gatewayState.getWidth() / 2, gatewayState.getY() + gatewayState.getHeight());
mxPoint eastPoint = new mxPoint(gatewayState.getX() + gatewayState.getWidth(), gatewayState.getY() + (gatewayState.getHeight() / 2);
mxPoint westPoint = new mxPoint(gatewayState.getX(), gatewayState.getY() + (gatewayState.getHeight() / 2);

double closestDistance = Double.MAX_VALUE;
mxPoint closestPoint = null;
for (mxPoint rhombusPoint : Arrays.asList(northPoint, southPoint, eastPoint, westPoint)) {
    double distance = euclidianDistance(startPoint, rhombusPoint);
    if (distance < closestDistance) {
        closestDistance = distance;
        closestPoint = rhombusPoint;
    }
}
startPoint.setX(closestPoint.getX());

// We also need a "NullPointerException" could be thrown; "closestPoint" is nullable here.
// Since we know the problem
// problem
if (points.size() > 0) {
    mxPoint nextPoint = null;
    nextPoint.setX(closestPoint.getX());
}

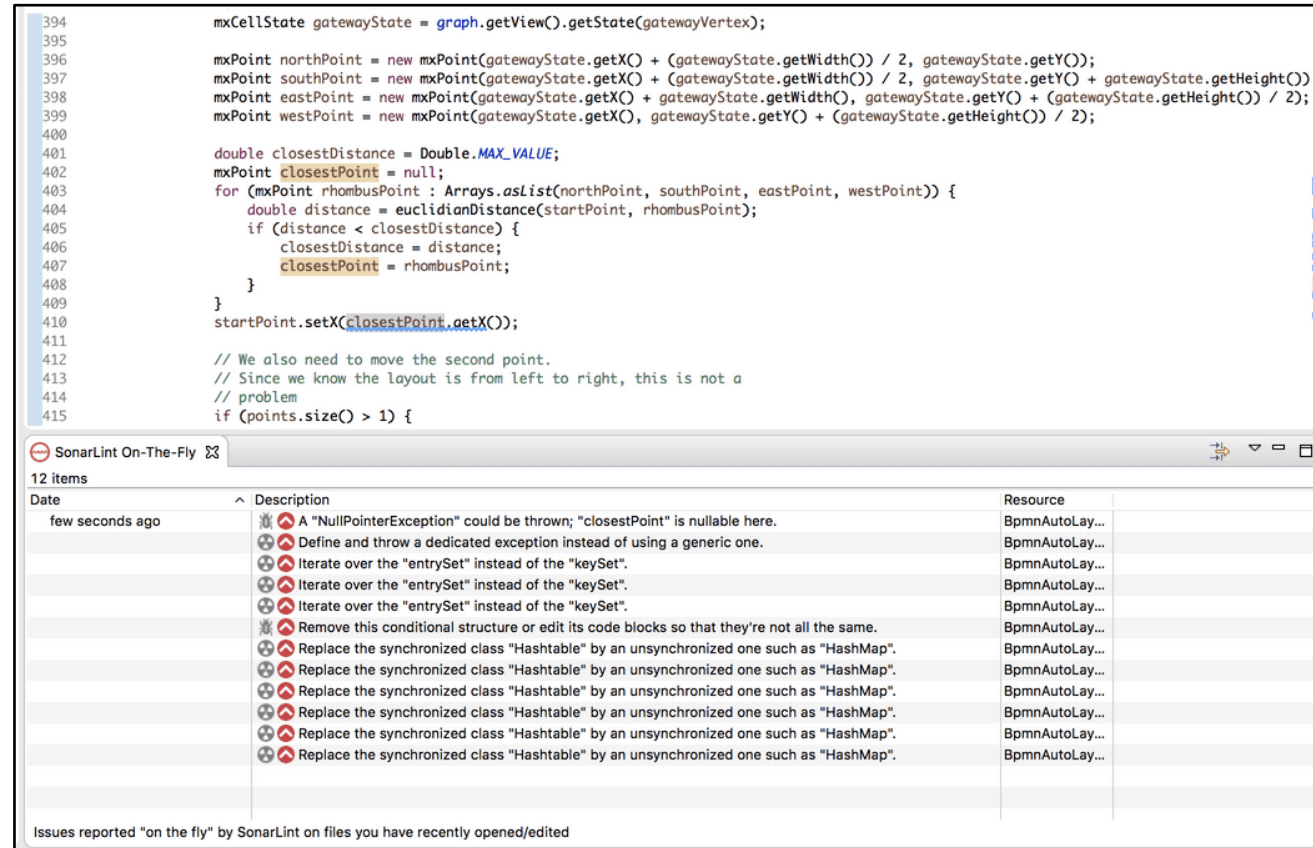
createDiagramInterchangeInformation(handledFlowElements.get(sequenceFlowId), optimizeEdgePoints(points));
}
```

3 quick fixes available:

- [Open description of rule squid:s2258](#)
- [Toggle all issue locations](#)
- [Deactivate rule squid:S2259](#)

Press "F2" for focus

SonarLint – Analizadores de código



The screenshot displays a code editor with Java code and a SonarLint On-The-Fly issues panel. The code defines a gateway state, calculates points, and finds the closest point to a start point. The issues panel lists 12 items, including a null pointer exception warning and several suggestions to replace synchronized classes with unsynchronized ones.

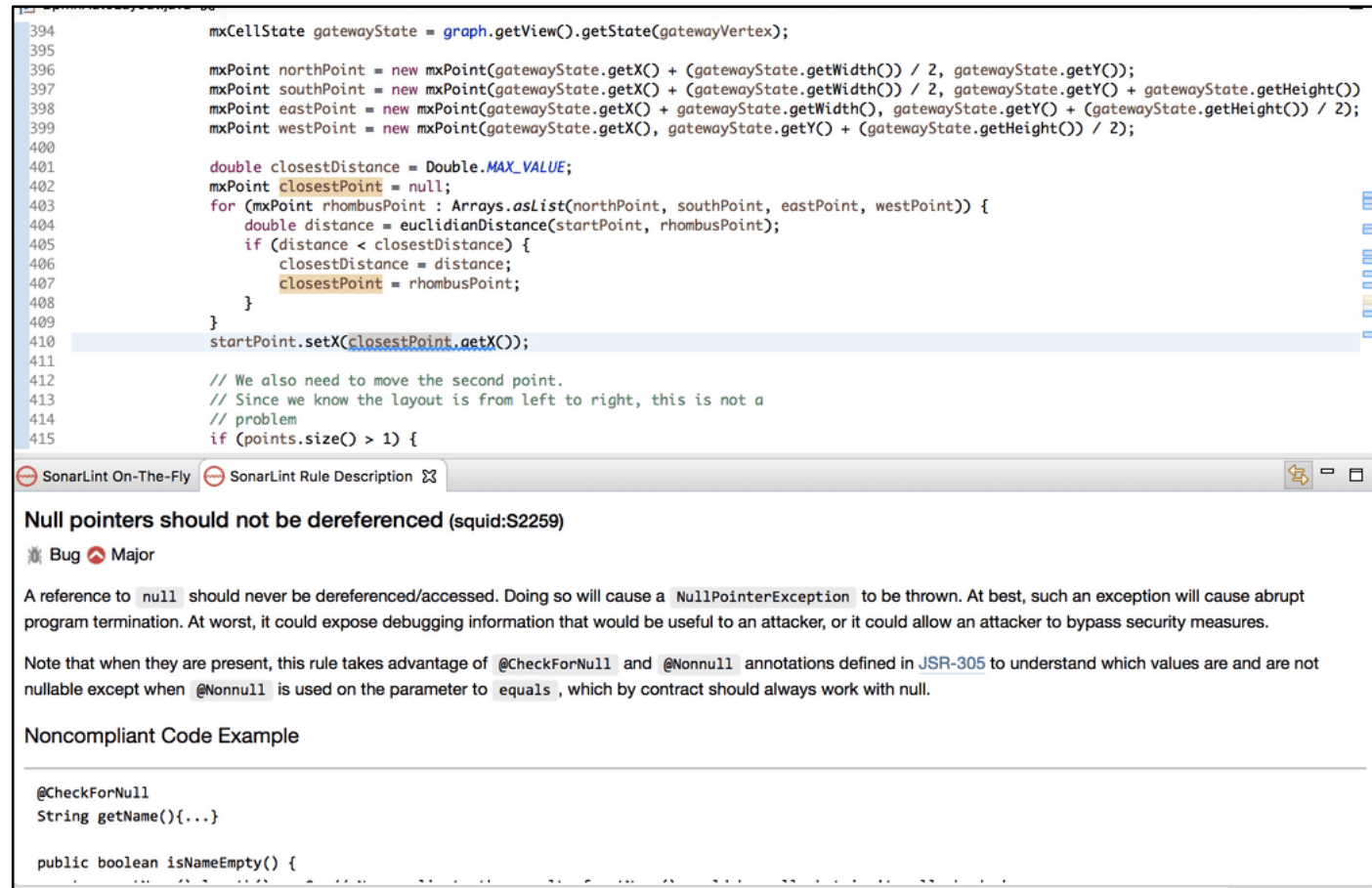
```
394      mxCellState gatewayState = graph.getView().getState(gatewayVertex);
395
396      mxPoint northPoint = new mxPoint(gatewayState.getX() + (gatewayState.getWidth()) / 2, gatewayState.getY());
397      mxPoint southPoint = new mxPoint(gatewayState.getX() + (gatewayState.getWidth()) / 2, gatewayState.getY() + gatewayState.getHeight());
398      mxPoint eastPoint = new mxPoint(gatewayState.getX() + gatewayState.getWidth(), gatewayState.getY() + (gatewayState.getHeight()) / 2);
399      mxPoint westPoint = new mxPoint(gatewayState.getX(), gatewayState.getY() + (gatewayState.getHeight()) / 2);
400
401      double closestDistance = Double.MAX_VALUE;
402      mxPoint closestPoint = null;
403      for (mxPoint rhombusPoint : Arrays.asList(northPoint, southPoint, eastPoint, westPoint)) {
404          double distance = euclidianDistance(startPoint, rhombusPoint);
405          if (distance < closestDistance) {
406              closestDistance = distance;
407              closestPoint = rhombusPoint;
408          }
409      }
410      startPoint.setX(closestPoint.getX());
411
412      // We also need to move the second point.
413      // Since we know the layout is from left to right, this is not a
414      // problem
415      if (points.size() > 1) {
```

SonarLint On-The-Fly 12 items



| Date | Description | Resource |
|-----------------|---|----------------|
| few seconds ago | ⚠ A "NullPointerException" could be thrown; "closestPoint" is nullable here. | BpmnAutoLay... |
| | ⚠ Define and throw a dedicated exception instead of using a generic one. | BpmnAutoLay... |
| | ⚠ Iterate over the "entrySet" instead of the "keySet". | BpmnAutoLay... |
| | ⚠ Iterate over the "entrySet" instead of the "keySet". | BpmnAutoLay... |
| | ⚠ Iterate over the "entrySet" instead of the "keySet". | BpmnAutoLay... |
| | ⚠ Remove this conditional structure or edit its code blocks so that they're not all the same. | BpmnAutoLay... |
| | ⚠ Replace the synchronized class "Hashtable" by an unsynchronized one such as "HashMap". | BpmnAutoLay... |
| | ⚠ Replace the synchronized class "Hashtable" by an unsynchronized one such as "HashMap". | BpmnAutoLay... |
| | ⚠ Replace the synchronized class "Hashtable" by an unsynchronized one such as "HashMap". | BpmnAutoLay... |
| | ⚠ Replace the synchronized class "Hashtable" by an unsynchronized one such as "HashMap". | BpmnAutoLay... |
| | ⚠ Replace the synchronized class "Hashtable" by an unsynchronized one such as "HashMap". | BpmnAutoLay... |
| | ⚠ Replace the synchronized class "Hashtable" by an unsynchronized one such as "HashMap". | BpmnAutoLay... |

Issues reported "on the fly" by SonarLint on files you have recently opened/edited



SonarLint – Analizadores de código



```
394      mxCellState gatewayState = graph.getView().getState(gatewayVertex);
395
396      mxPoint northPoint = new mxPoint(gatewayState.getX() + (gatewayState.getWidth() / 2, gatewayState.getY());
397      mxPoint southPoint = new mxPoint(gatewayState.getX() + (gatewayState.getWidth() / 2, gatewayState.getY() + gatewayState.getHeight());
398      mxPoint eastPoint = new mxPoint(gatewayState.getX() + gatewayState.getWidth(), gatewayState.getY() + (gatewayState.getHeight() / 2);
399      mxPoint westPoint = new mxPoint(gatewayState.getX(), gatewayState.getY() + (gatewayState.getHeight() / 2);
400
401      double closestDistance = Double.MAX_VALUE;
402      mxPoint closestPoint = null;
403      for (mxPoint rhombusPoint : Arrays.asList(northPoint, southPoint, eastPoint, westPoint)) {
404          double distance = euclidianDistance(startPoint, rhombusPoint);
405          if (distance < closestDistance) {
406              closestDistance = distance;
407              closestPoint = rhombusPoint;
408          }
409      }
410      startPoint.setX(closestPoint.getX());
411
412      // We also need to move the second point.
413      // Since we know the layout is from left to right, this is not a
414      // problem
415      if (points.size() > 1) {
```

SonarLint On-The-Fly SonarLint Rule Description  

Null pointers should not be dereferenced (squid:S2259)

 Bug  Major

A reference to `null` should never be dereferenced/accessed. Doing so will cause a `NullPointerException` to be thrown. At best, such an exception will cause abrupt program termination. At worst, it could expose debugging information that would be useful to an attacker, or it could allow an attacker to bypass security measures.

Note that when they are present, this rule takes advantage of `@CheckForNull` and `@Nonnull` annotations defined in [JSR-305](#) to understand which values are and are not nullable except when `@Nonnull` is used on the parameter to `equals`, which by contract should always work with `null`.

Noncompliant Code Example

```
@CheckForNull
String getName(){...}

public boolean isEmpty() {
```

Actividad

Elige un analizador de código (por ejemplo, uno de los tres anteriores, u otro diferente). Elabora un manual de instalación del plugin, Pruébalo y muestra alguna de sus funcionalidades. Recuerda realizar capturas de pantalla durante el proceso.

Documentación

Documentación

La documentación de un problema consta de las descripciones de los pasos a dar en el proceso de resolución de dicho problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final. Programas deficientemente documentados son difíciles de leer, más difíciles de depurar y casi imposibles de mantener y modificar.

Documentación

La documentación de un programa puede ser **interna** y **externa**.

La documentación interna es la contenida en líneas de comentarios.

La documentación externa incluye análisis, diagramas de flujo y/o pseudocódigos, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

Documentación Externa

Como ya vimos por encima cuando hablamos del ciclo de vida del software, en la documentación formal de un proyecto vienen recogidos los siguientes documentos:

Documentación Externa

- ❖ **Oferta de desarrollo.** Recoge una descripción del acuerdo alcanzado por la empresa proveedora y la empresa cliente. Deben especificarse valoraciones económicas, valoraciones temporales, disposiciones técnicas y una referencia al documento de especificaciones de requisitos.
- ❖ **Documentación de las especificaciones.** Recoge qué es lo que hay que hacer, con qué se debe hacer y cómo hacerlo.

Documentación Externa

- ❖ **Documentación del desarrollo.** Recoge toda la información necesaria que se va generando durante el desarrollo de la aplicación. No se entrega al cliente.
- ❖ **Documentación de las pruebas.** Recoge todos los casos de prueba que se han generado para toda la aplicación. Se entrega al cliente cuando finaliza el desarrollo de la aplicación y deberá contener los estados del control de cambios (solucionadas, en observación, rechazadas y motivo).

Documentación Externa

Documentación del cliente. Deberá entregarse:

- ❖ **Manual de despliegue.** Recoge la información necesaria para poner en explotación la aplicación. Va dirigido fundamentalmente a la instalación y configuración.
- ❖ **Manual de administración.** Es el documento que recoge las tareas que deben realizarse para el mantenimiento y administración de aplicación.
- ❖ **Manual de usuario.** Reúne la información, normas y documentación necesaria para que el usuario conozca y utilice adecuadamente la aplicación desarrollada. Los objetivos que se persiguen son el correcto uso de la aplicación y que permita detectar y corregir errores.

Documentación Externa – Pruebas

La documentación de las pruebas es necesaria para una buena organización de las mismas, así como para asegurar su reutilización. También contribuye a la eficacia y eficiencia de la aplicación generada.

Los siguientes documentos están asociados a la fase de diseño de las pruebas:

Documentación Externa – Pruebas

❖ **Plan de pruebas.** El objetivo del documento es señalar el enfoque, los recursos y el esquema de actividades de prueba, así como los elementos a probar, las características, los criterios, las actividades de prueba, el personal responsable y los riesgos asociados.

Documentación Externa – Pruebas

- ❖ **Especificación del diseño de pruebas.** El objetivo del documento es identificar las características de los elementos software que se deben probar y los criterios de paso/fallo de la prueba.
- ❖ **Especificación de caso de prueba.** El objetivo del documento es definir uno de los casos de prueba identificado por una especificación del diseño de las pruebas, detallando los elementos software a probar, las entradas y salidas del caso de prueba, y los requisitos especiales del procedimiento de prueba.

Documentación Externa – Pruebas

❖ **Especificación de procedimiento de prueba.** El objetivo del documento es especificar las acciones necesarias para la ejecución de un conjunto de casos de prueba o, más generalmente, los pasos utilizados para analizar un elemento software con el propósito de evaluar un conjunto de características del mismo.

Documentación Externa – Pruebas

Los siguientes documentos están asociados a la fase de ejecución de las pruebas:

- ❖ **Histórico de pruebas.** El objetivo del documento es documentar todos los hechos relevantes ocurridos durante la ejecución de las pruebas.
- ❖ **Informe de incidente.** El objetivo del documento es documentar cada incidente (por ejemplo, una interrupción en las pruebas debido a un corte de electricidad, bloqueo del teclado, etc.) ocurrido en la prueba y que requiera una posterior investigación.
- ❖ **Informe resumen de pruebas.** El objetivo del documento es resumir los resultados de las actividades de prueba (las señaladas en el propio informe) y aportar una evaluación del software basada en dichos resultados.

Documentación Interna

- ❖ El proceso de documentación de código, es uno de los aspectos más importantes de la labor de un programador.
- ❖ Documentar el código nos sirve para explicar su funcionamiento, punto por punto, de forma que cualquier persona que lea el comentario, puede entender la finalidad del código.

Documentación Interna

- ❖ La labor de documentación es fundamental para la detección de errores y para su mantenimiento posterior, que en muchos casos, es realizado por personas diferentes a las que intervinieron en su creación.
- ❖ Hay que tener en cuenta que todos los programas tienen errores y todos los programas sufren modificaciones a lo largo de su vida.

Documentación Interna

- ❖ La documentación añade explicaciones de la función del código, de las características de un método, etc. Debe tratar de explicar todo lo que no resulta evidente. Su objetivo no es repetir lo que hace el código, sino explicar por qué se hace.
- ❖ La documentación explicará cual es la finalidad de un clase, de un paquete, qué hace un método, para que sirve una variable, qué se espera del uso de una variable, qué algoritmo se usa, por qué hemos implementado de una manera y de otro, qué se podría mejorar en el futuro, etc.

Uso de comentarios

- ❖ Uno de los elementos básicos para documentar código, es el uso de comentarios.
- ❖ Un comentario es una anotación que se realiza en el código, pero que el compilador va a ignorar.
- ❖ Sirve para indicar a los desarrolladores de código diferentes aspectos del código que pueden ser útiles.

Uso de comentarios

En principio, los comentarios tienen dos propósitos diferentes:

- ❖ Explicar el objetivo de las sentencias. De forma que el programador sepa en todo momento la función de esa sentencia, tanto si lo diseñó el mismo como si son otros los que quieren entenderlo o modificarlo.
- ❖ Explicar qué realiza un método, o clase, no cómo lo realiza. En este caso, se trata de explicar los valores que va a devolver un método, pero no se trata de explicar cómo se ha diseñado.

Uso de comentarios

- ❖ En el caso del lenguaje Java, C# y C, los comentarios, se implementan de forma similar.
- ❖ Cuando se trata de explicar la función de una sentencia, se usan los caracteres `//` seguidos del comentario, o con los caracteres `/*` y `*/`, situando el comentario entre ellos: `/* comentario */`

Uso de comentarios

- ❖ Otro tipo de comentarios que se utilizan en Java, son los que se utilizan para explicar qué hace un código, se denominan comentarios **JavaDoc** y se escriben empezando por **/**** y terminando con ***/** , estos comentarios pueden ocupar varias líneas.
- ❖ Este tipo de comentarios tienen que seguir una estructura prefijada.

Uso de comentarios

- ❖ Los comentarios son obligatorios con JavaDoc, y se deben incorporar al principio de cada clase, al principio de cada método y al principio de cada variable de clase. No es obligatorio, pero en muchas situaciones es conveniente, poner los comentarios al principio de un fragmento de código que no resulta lo suficientemente claro, a la largo de bucles, o si hay alguna línea de código que no resulta evidente y pueda llevarnos a confusión.
- ❖ **Hay que tener en cuenta, que, si el código es modificado, también se deberán modificar los comentarios.**

Alternativas

- ❖ En la actualidad, el desarrollo rápido de aplicaciones, en muchos casos, va en detrimento de una buena documentación del código.
- ❖ Si el código no está documentado, puede resultar bastante difícil de entender, y por tanto de solucionar errores y de mantenerlo.

Alternativas

- ❖ La primera alternativa que surge para documentar código, son los comentarios. Con los comentarios, documentamos la funcionalidad de una línea de código, de un método o el comportamiento de una determinada clase.
- ❖ Existen diferentes herramientas que permiten automatizar, completar y enriquecer nuestra documentación.

Alternativas

- ❖ Podemos citar JavaDoc, SchemeSpy y Doxygen, que producen una documentación actualizada, precisa y utilizable en línea, incluyendo además, con SchemeSpy y Doxygen, modelos de bases de datos gráficos y diagramas.
- ❖ **Insertando comentario en el código más difícil de entender, y utilizando la documentación generada por alguna de las herramientas citadas anteriormente, se genera la suficiente información para ayudar a cualquier nuevo programador o programadora.**

Documentación de clases

- ❖ Las clases que se implementan en una aplicación, deben de incluir comentarios. Al utilizar un entorno de programación para la implementación de la clase, debemos seguir una serie de pautas, muchas de las cuales las realiza el IDE de forma transparente, en el diseño y documentación del código.
- ❖ Cuando se implementa una clase, se deben incluir comentarios. En el lenguaje Java, los criterios de documentación de clases, son los establecidos por JavaDoc.

Documentación de clases

❖ Los comentarios de una clase deben comenzar con `/**` y terminar con `*/`. Entre la información que debe incluir un comentario de clase debe incluirse, al menos las etiquetas **@autor** y **@version**, donde **@autor** identifica el nombre del autor o autora de la clase y **@version**, la identificación de la versión y fecha.

Documentación de clases

- ❖ Dentro de la la clase, también se documentan los constructores y los métodos. Al menos se indican las etiquetas:
- ❖ **@param**: seguido del nombre, se usa para indicar cada uno de los parámetros que tienen el constructor o método.
- ❖ **@return**: si el método no es void, se indica lo que devuelve.
- ❖ **@exception**: se indica el nombre de la excepción, especificando cuales pueden lanzarse.
- ❖ **@throws**: se indica el nombre de la excepción, especificando las excepciones que pueden lanzarse.

Herramientas

- ❖ Los entornos de programación que implementa Java, como Eclipse o Netbeans, incluyen una herramienta que va a generar páginas HTML de documentación a partir de los comentarios incluidos en el código fuente. La herramienta ya se ha indicado en los puntos anteriores, y es JavaDoc.

Herramientas

- ❖ Para que JavaDoc pueda generar las páginas HTML es necesario seguir una serie de normas de documentación en el código fuente, estas son:
- ❖ Los comentarios JavaDoc deben empezar por `/**` y terminar por `*/`.
- ❖ Los comentarios pueden ser a nivel de clase, a nivel de variable y a nivel de método.
- ❖ La documentación se genera para métodos `public` y `protected`.

Herramientas

❖ Se puede usar tag para documentar diferentes aspectos determinados del código, como parámetros. Los tags más habituales son los siguientes:

| Tipo de tag | Formato | Descripción |
|----------------|------------------------|---|
| Todos. | <code>@see.</code> | Permite crear una referencia a la documentación de otra clase o método. |
| Clases. | <code>@version.</code> | Comentario con datos indicativos del número de versión. |
| Clases. | <code>@author.</code> | Nombre del autor. |
| Clases. | <code>@since.</code> | Fecha desde la que está presente la clase. |

Herramientas

❖ Se puede usar tag para documentar diferentes aspectos determinados del código, como parámetros. Los tags más habituales son los siguientes:

| | | |
|-----------------|---------------------|---|
| Métodos. | @param. | Parámetros que recibe el método. |
| Métodos. | @return. | Significado del dato devuelto por el método |
| Métodos. | @throws. | Comentario sobre las excepciones que lanza. |
| Métodos. | @deprecated. | Indicación de que el método es obsoleto. |