|  |
| --- |
| megachat  Version 0.0.1-SNAPSHOT  Code analysis |

|  |
| --- |
| **By: Administrator**  **2023-06-16** |

# Content

[Content 1](#_Toc97156258)

[Introduction 2](#_Toc97156259)

[Configuration 2](#_Toc97156260)

[Synthesis 3](#_Toc97156261)

[Analysis Status 3](#_Toc97156262)

[Quality gate status 3](#_Toc97156263)

[Metrics 3](#_Toc97156264)

[Tests 3](#_Toc97156265)

[Detailed technical debt 3](#_Toc97156266)

[Metrics Range 5](#_Toc97156267)

[Volume 5](#_Toc97156268)

[Issues 6](#_Toc97156269)

[Charts 6](#_Toc97156270)

[Issues count by severity and type 8](#_Toc97156271)

[Issues List 8](#_Toc97156272)

[Security Hotspots 9](#_Toc97156273)

[Security hotspots count by category and priority 9](#_Toc97156274)

[Security hotspots List 9](#_Toc97156275)

# Introduction

This document contains results of the code analysis of megachat.

Chat application

# Configuration

* Quality Profiles
  + Names: Sonar way [Java]; Sonar way [XML];
  + Files: AYjEI4WgUMY6zXrqjMqW.json; AYjEI4aqUMY6zXrqjM2U.json;
* Quality Gate
  + Name: Sonar way
  + File: Sonar way.xml

# Synthesis

## Analysis Status

|  |  |  |  |
| --- | --- | --- | --- |
| Reliability | Security | Security Review | Maintainability |
| C.png | **A.png** | **A.png** | **A.png** |

## Quality gate status

|  |  |
| --- | --- |
| Quality Gate Status | **OK.png** |



## Metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Coverage | Duplication | Comment  density | Median number of lines of code per file | Adherence to coding standard |
| 0.0 % | **0.0 %** | **1.3 %** | **29.0** | **99.8 %** |

## Tests

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Total | Success Rate | Skipped | Errors | Failures |
| 1 | **100.0 %** | **0** | **0** | **0** |

## Detailed technical debt

|  |  |  |  |
| --- | --- | --- | --- |
| Reliability | Security | Maintainability | Total |
| 0d 0h 20min | - | 0d 0h 25min | 0d 0h 45min |

## Metrics Range

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Cyclomatic  Complexity | Cognitive  Complexity | Lines of code per file | Comment  density (%) | Coverage | Duplication (%) |
| Min | 1.0 | 0.0 | 9.0 | 0.0 | 0.0 | 0.0 |
| Max | 14.0 | 1.0 | 110.0 | 4.3 | 0.0 | 0.0 |

## Volume

|  |  |
| --- | --- |
| Language | Number |
| Java | 110 |
| XML | 38 |
| Total | 148 |

# Issues

## Charts

## Issues count by severity and type

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type / Severity | INFO | MINOR | MAJOR | CRITICAL | BLOCKER |
| BUG | 0 | 0 | 2 | 0 | 0 |
| VULNERABILITY | 0 | 0 | 0 | 0 | 0 |
| CODE\_SMELL | 0 | 0 | 2 | 0 | 1 |

## Issues List

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Type | Severity | Number |
| Null pointers should not be dereferenced | 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 isNameEmpty() { return getName().length() == 0; // Noncompliant; the result of getName() could be null, but isn't null-checked } Connection conn = null; Statement stmt = null; try{ conn = DriverManager.getConnection(DB\_URL,USER,PASS); stmt = conn.createStatement(); // ... }catch(Exception e){ e.printStackTrace(); }finally{ stmt.close(); // Noncompliant; stmt could be null if an exception was thrown in the try{} block conn.close(); // Noncompliant; conn could be null if an exception was thrown } private void merge(@Nonnull Color firstColor, @Nonnull Color secondColor){...} public void append(@CheckForNull Color color) { merge(currentColor, color); // Noncompliant; color should be null-checked because merge(...) doesn't accept nullable parameters } void paint(Color color) { if(color == null) { System.out.println("Unable to apply color " + color.toString()); // Noncompliant; NullPointerException will be thrown return; } ... } See MITRE, CWE-476 - NULL Pointer Dereference CERT, EXP34-C. - Do not dereference null pointers CERT, EXP01-J. - Do not use a null in a case where an object is required | BUG | MAJOR | 2 |
| Tests should include assertions |  | CODE\_SMELL | BLOCKER | 1 |
| "Preconditions" and logging arguments should not require evaluation | Passing message arguments that require further evaluation into a Guava com.google.common.base.Preconditions check can result in a performance penalty. That’s because whether or not they’re needed, each argument must be resolved before the method is actually called. Similarly, passing concatenated strings into a logging method can also incur a needless performance hit because the concatenation will be performed every time the method is called, whether or not the log level is low enough to show the message. Instead, you should structure your code to pass static or pre-computed values into Preconditions conditions check and logging calls. Specifically, the built-in string formatting should be used instead of string concatenation, and if the message is the result of a method call, then Preconditions should be skipped altogether, and the relevant exception should be conditionally thrown instead. Noncompliant Code Example logger.log(Level.DEBUG, "Something went wrong: " + message); // Noncompliant; string concatenation performed even when log level too high to show DEBUG messages logger.fine("An exception occurred with message: " + message); // Noncompliant LOG.error("Unable to open file " + csvPath, e); // Noncompliant Preconditions.checkState(a &gt; 0, "Arg must be positive, but got " + a); // Noncompliant. String concatenation performed even when a &gt; 0 Preconditions.checkState(condition, formatMessage()); // Noncompliant. formatMessage() invoked regardless of condition Preconditions.checkState(condition, "message: %s", formatMessage()); // Noncompliant Compliant Solution logger.log(Level.SEVERE, "Something went wrong: {0} ", message); // String formatting only applied if needed logger.fine("An exception occurred with message: {}", message); // SLF4J, Log4j logger.log(Level.SEVERE, () -&gt; "Something went wrong: " + message); // since Java 8, we can use Supplier , which will be evaluated lazily LOG.error("Unable to open file {0}", csvPath, e); if (LOG.isDebugEnabled()) { LOG.debug("Unable to open file " + csvPath, e); // this is compliant, because it will not evaluate if log level is above debug. } Preconditions.checkState(arg &gt; 0, "Arg must be positive, but got %d", a); // String formatting only applied if needed if (!condition) { throw new IllegalStateException(formatMessage()); // formatMessage() only invoked conditionally } if (!condition) { throw new IllegalStateException("message: " + formatMessage()); } Exceptions catch blocks are ignored, because the performance penalty is unimportant on exceptional paths (catch block should not be a part of standard program flow). Getters are ignored as well as methods called on annotations which can be considered as getters. This rule accounts for explicit test-level testing with SLF4J methods isXXXEnabled and ignores the bodies of such if statements. | CODE\_SMELL | MAJOR | 1 |
| Printf-style format strings should be used correctly | Because printf-style format strings are interpreted at runtime, rather than validated by the compiler, they can contain errors that result in the wrong strings being created. This rule statically validates the correlation of printf-style format strings to their arguments when calling the format(...) methods of java.util.Formatter, java.lang.String, java.io.PrintStream, MessageFormat, and java.io.PrintWriter classes and the printf(...) methods of java.io.PrintStream or java.io.PrintWriter classes. Noncompliant Code Example String.format("First {0} and then {1}", "foo", "bar"); //Noncompliant. Looks like there is a confusion with the use of {{java.text.MessageFormat}}, parameters "foo" and "bar" will be simply ignored here String.format("Display %3$d and then %d", 1, 2, 3); //Noncompliant; the second argument '2' is unused String.format("Too many arguments %d and %d", 1, 2, 3); //Noncompliant; the third argument '3' is unused String.format("First Line\n"); //Noncompliant; %n should be used in place of \n to produce the platform-specific line separator String.format("Is myObject null ? %b", myObject); //Noncompliant; when a non-boolean argument is formatted with %b, it prints true for any nonnull value, and false for null. Even if intended, this is misleading. It's better to directly inject the boolean value (myObject == null in this case) String.format("value is " + value); // Noncompliant String s = String.format("string without arguments"); // Noncompliant MessageFormat.format("Result '{0}'.", value); // Noncompliant; String contains no format specifiers. (quote are discarding format specifiers) MessageFormat.format("Result {0}.", value, value); // Noncompliant; 2nd argument is not used MessageFormat.format("Result {0}.", myObject.toString()); // Noncompliant; no need to call toString() on objects java.util.Logger logger; logger.log(java.util.logging.Level.SEVERE, "Result {0}.", myObject.toString()); // Noncompliant; no need to call toString() on objects logger.log(java.util.logging.Level.SEVERE, "Result.", new Exception()); // compliant, parameter is an exception logger.log(java.util.logging.Level.SEVERE, "Result '{0}'", 14); // Noncompliant - String contains no format specifiers. logger.log(java.util.logging.Level.SEVERE, "Result " + param, exception); // Noncompliant; Lambda should be used to differ string concatenation. org.slf4j.Logger slf4jLog; org.slf4j.Marker marker; slf4jLog.debug(marker, "message {}"); slf4jLog.debug(marker, "message", 1); // Noncompliant - String contains no format specifiers. org.apache.logging.log4j.Logger log4jLog; log4jLog.debug("message", 1); // Noncompliant - String contains no format specifiers. Compliant Solution String.format("First %s and then %s", "foo", "bar"); String.format("Display %2$d and then %d", 1, 3); String.format("Too many arguments %d %d", 1, 2); String.format("First Line%n"); String.format("Is myObject null ? %b", myObject == null); String.format("value is %d", value); String s = "string without arguments"; MessageFormat.format("Result {0}.", value); MessageFormat.format("Result '{0}' = {0}", value); MessageFormat.format("Result {0}.", myObject); java.util.Logger logger; logger.log(java.util.logging.Level.SEVERE, "Result {0}.", myObject); logger.log(java.util.logging.Level.SEVERE, "Result {0}'", 14); logger.log(java.util.logging.Level.SEVERE, exception, () -&gt; "Result " + param); org.slf4j.Logger slf4jLog; org.slf4j.Marker marker; slf4jLog.debug(marker, "message {}"); slf4jLog.debug(marker, "message {}", 1); org.apache.logging.log4j.Logger log4jLog; log4jLog.debug("message {}", 1); See CERT, FIO47-C. - Use valid format strings | CODE\_SMELL | MAJOR | 1 |

# Security Hotspots

## Security hotspots count by category and priority

|  |  |  |  |
| --- | --- | --- | --- |
| Category / Priority | LOW | MEDIUM | HIGH |
| LDAP Injection | 0 | 0 | 0 |
| Object Injection | 0 | 0 | 0 |
| Server-Side Request Forgery (SSRF) | 0 | 0 | 0 |
| XML External Entity (XXE) | 0 | 0 | 0 |
| Insecure Configuration | 0 | 0 | 0 |
| XPath Injection | 0 | 0 | 0 |
| Authentication | 0 | 0 | 0 |
| Weak Cryptography | 0 | 0 | 0 |
| Denial of Service (DoS) | 0 | 0 | 0 |
| Log Injection | 0 | 0 | 0 |
| Cross-Site Request Forgery (CSRF) | 0 | 0 | 0 |
| Open Redirect | 0 | 0 | 0 |
| Permission | 0 | 0 | 0 |
| SQL Injection | 0 | 0 | 0 |
| Encryption of Sensitive Data | 0 | 0 | 0 |
| Traceability | 0 | 0 | 0 |
| Buffer Overflow | 0 | 0 | 0 |
| File Manipulation | 0 | 0 | 0 |
| Code Injection (RCE) | 0 | 0 | 0 |
| Cross-Site Scripting (XSS) | 0 | 0 | 0 |
| Command Injection | 0 | 0 | 0 |
| Path Traversal Injection | 0 | 0 | 0 |
| HTTP Response Splitting | 0 | 0 | 0 |
| Others | 0 | 0 | 0 |

## Security hotspots List