Report for Chessboard Validation Moves

Static Analysis Tool:
SonarQube, Sonar Lint.

I began by installing Maven on my system, enabling smooth integration of SonarQube and Sonar Lint. Configured SonarQube with Maven, setting up foundational parameters for static analysis. Utilized Sonar Lint within the IDE for real-time code examination and refinement. Restructured the codebase based on insights from SonarQube and Sonar Lint, ensuring adherence to best practices. This proactive approach laid a solid groundwork for subsequent test case development and ensured a robust Chessboard Validation project.

Code Coverage Tool:

Jacoco, IntelliJ

For code coverage analysis, I used both IntelliJ's built-in tool and Jacoco. Utilizing IntelliJ's native functionality, I easily assessed code coverage within the IDE. Additionally, I integrated Jacoco into the project by adding its dependency and properties to the pom.xml file.

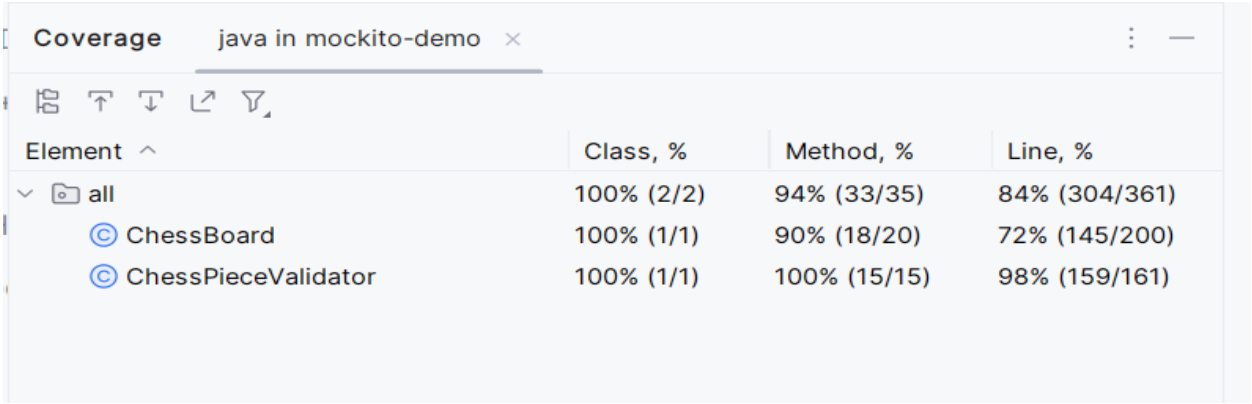
While reviewing the code, I did encounter some instances of code smells, particularly within return and print statements. However, considering their non-critical nature, I chose to prioritize other aspects of the project.

Below table is the code coverage report of Jacoco and IntelliJ

Jacoco: Programming Project\Final Chess Board\Final Chess Board 2\target\site\jacoco\index.html
default

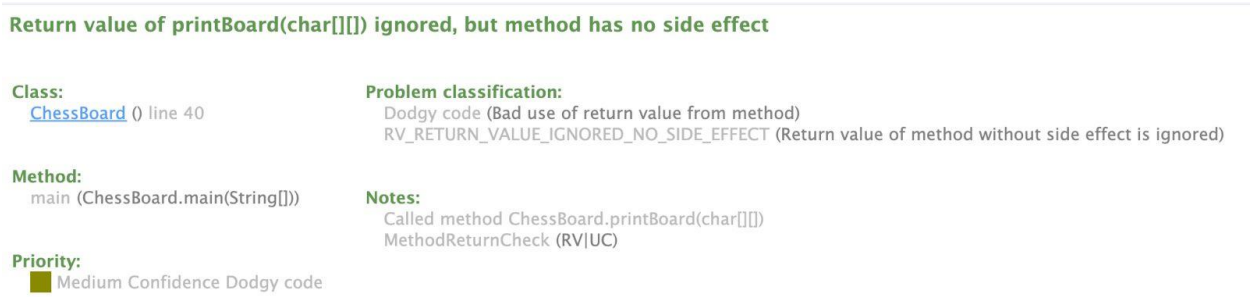
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ChessBoard		78%		72%	40	120	56	203	2	21	0	1
ChessPieceValidator		98%		92%	15	128	3	163	1	16	0	1
Total	254 of 2,351	89%	67 of 406	83%	55	248	59	366	3	37	0	2

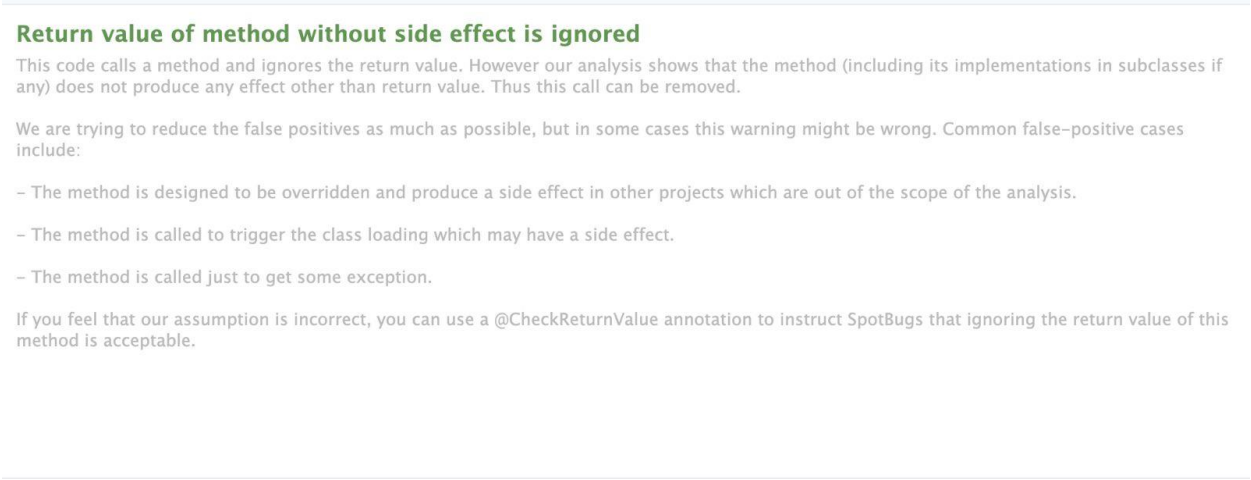
IntelliJ:



Bugs:

SpotBugs:
When addressing bugs within the project, I employed the SpotBugs plugin integrated into IntelliJ for thorough bug detection. To enhance code quality, I diligently ran each test case and meticulously examined the results for any potential bugs. While the process identified minor issues, such as a method calling error resulting. I've included a screenshot showcasing the encountered bug for reference.





Unfortunately, I encountered difficulty in crafting a test case specifically for user inputs due to the involvement of Streams or Buffered Reader. However, I want to make sure that I have covered all other aspects within the test cases.

Steps I Took to Improve Code Coverage:

Despite initial efforts, achieving maximum code coverage proved challenging. Integration of JaCoCo proved instrumental in this regard. Regularly examining JaCoCo reports post-build allowed me to identify uncovered lines and branches in the codebase. Subsequently, I tailored my test cases to address these gaps. This approach, coupled with leveraging IntelliJ's tools, facilitated a comprehensive examination of the codebase. Though challenges persisted, particularly in testing user input scenarios, I remained proactive in refining test cases to enhance overall coverage. Additionally, I employed SonarQube and Sonar Lint for static analysis, further refining the codebase.