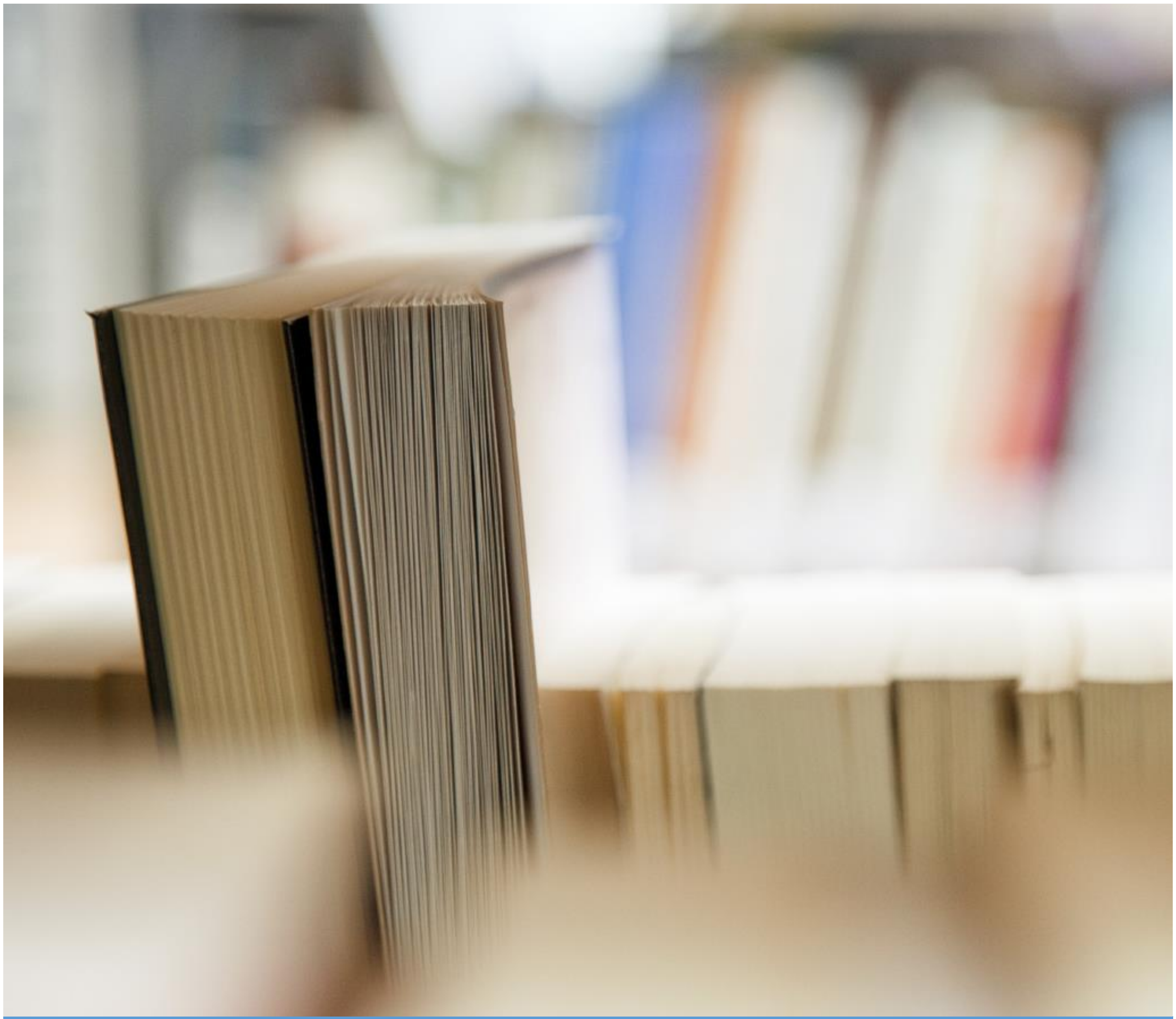


Comparison Of Static Code Analysis Tools For JAVA

Frank Lin, Prateek Tandon

fjlin, ptandon1

5/9/14

17-691 SOFTWARE DEVELOPMENT TOOLS

Abstract

Static code analysis is an essential process in any software development process. The development of automated tools has made the process of static code analysis more efficient and effective. In this paper, we compare, contrast, and analyze 4 popular open source static analysis tools (FindBugs, CheckStyle, PMD, and SonarGragh). We first give a brief overview about how each of these tools work and then test code using these tools and compare the results with each other.

Introduction

The use of different methods to review source code has become an absolute necessity in the software development process. Static code analysis is the analysis of computer code that is performed without the actual execution of the code to find potential bugs, vulnerabilities, structural issues, and assure overall code quality. Static analysis tools look for erroneous patterns using a defined set of rules to comprehensively analyze the source code. Most static analysis is done through the use of automated tools or manual inspection of code. The main goal of static code analysis is to detect common errors in the development of code before it is too far in the development process. Not only can this save time and money in the development process, it can also lead to an overall better and more efficient code.

Static code analysis tools in JAVA operate in one of two ways, they either work directly on the source code or on the compiled byte code. In general, most static analysis tools read the code, construct some model, and match error patterns to a set of predefined rules. These tools can also perform some data-flow analysis, which is important in many cases.

Though there are many benefits of using static code analysis, there are also downsides that should not be overlooked when using such analysis. Let's examine some of the pros and cons of static code analysis.

Pros

Static analysis provides a thorough analysis of code without actual execution. It does this by scanning through all the code and checking for errors. As a result, the analysis can be applied to incomplete or incorrect code and before the development of test cases. This allows for earlier analysis of code in the development process. Also, static analysis is independent of the run time environment and can locate errors in exception handling and logging. If there are any issues present in code that is unused, static analysis is able to detect the code.

Static analysis is highly extensible in that the user can define project specific rules. This makes different tools highly customizable to different projects and allows for the widespread reuse of tools. By using static code tools, developers are able to efficiently find bugs early in the development cycle, thus resulting in a faster and cheaper fix turnaround time. As a result, there is an overall reduction of time and costs when maintaining and fixing code.

Overall, this form of analysis can be more efficient and cost less time and resources when compared to other methods. It can help catch subtle errors that can otherwise be missed by other tests; such as copy and pasted methods and null pointers. Static analysis tools can pinpoint the exact location of the error and alert the user of a multitude of different potentially erroneous code. Static analysis can point out unclear code or code with bad structure that can result in confusion amongst programmers. It can help identify and verify different execution paths that other methods fail to cover. This can greatly improve the overall quality of code.

Cons

Though there are many benefits of using static analysis, there are also big shortcomings that prevent this form of analysis from being ideal for every situation. A major problem of static analysis is that it results in a high number of false positive. The appearances of false negative in the analysis create a false sense of security for the developers. Though this error is prevalent with any tool, it is especially prevalent in static code analysis. Since the method goes through 100% of the source code, it results in many warnings that are non-issues and are safe to ignore. The false positives result in many problems for developers. They have to determine what issues are safe to ignore and what issues are actually detrimental to the code. The number of false positives can be reduced by implementing additional rules that the analysis tools follow and make them more specific to the project.

Static analysis tools are not one-and-done tools; they should be continuously used throughout the development process. In some cases, static analysis can take too long to run and as a result, developers give up using them. This is why many code analysis processes are integrated as part of the build process instead of being optional.

Lastly, there isn’t a universally accepted set of rules for static analysis. How effective a tool is completely depends on the set of rules implemented. The stronger and more comprehensive the rules are, the more efficient and effective the overall test is. With an increase in the comprehensiveness, there is also an increase in the overall run time and complexity. Common static analysis tools cannot detect conditional errors and static analysis tools cannot detect errors resulting from the runtime environment. This requires additional testing using dynamic code analysis.

Project Overview

Using analysis tools for static code analysis is much more favorable than manually checking code due the increase in speed. By using tools, code can be evaluated faster and much less formal training is needed for effective analysis of code. Any trained programmer can effectively apply the tools without completely understand the workings of the code. Though the output of these tools still requires human evaluation, using tools is still a much better alternative.

Static analysis tools can detect some of the most obscure problems in places that are hard to reach for other forms of analysis. A good static analysis tool must be fast, efficient, and easy to use. Developers that might not know a great deal about certain issues must be able to understand how to work the tools and interpret the results. Static analysis tools cannot solve all problems in a code due to the nature of how they work. Though more advanced tools are being developed, we are still a long way from the creation of a universally accepted set of rules for a universally accepted tool.

In our project, we selected 4 popular open source tools for static code analysis. The 4 tools are FindBugs, CheckStyle, PMD, and SonarGraph. We selected these tools based on their download popularity and through web searches. In Section 1, we give a brief overview of these tools and introduce how they work, their rules, and other important properties. In the Section 2, we use these tools to analyze code and compare their results with one another. Lastly, in section 3, we discuss our findings and give suggestions for future studies.

Section 1

FindBugs

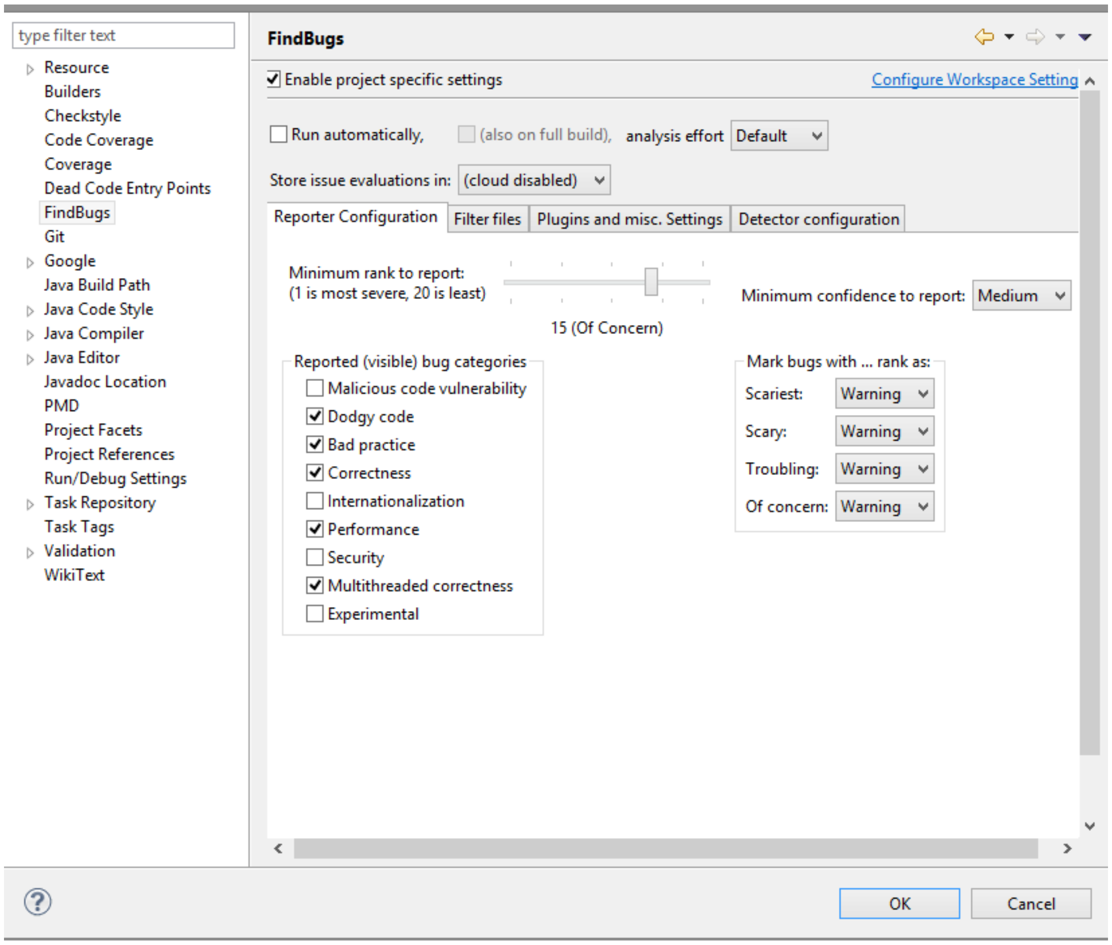
FindBugs is a popular open source tool used for static analysis of Java code. It was created by Bill Pugh and David Hovemeyer at the University of Maryland. It is written in JAVA and is under the Lesser GNU General public license. Unlike other static analysis tools, FindBugs operates on byte code rather than source code and focuses on finding real bugs and potential performance problems instead of just style and formatting issues. The software is distributed as a stand-alone GUI or as a plugin for Eclipse, NetBeans, IntelliJ IDEA, Gradle, Hudson, and Jenkins.

FindBugs operates by looking for instances of code patterns that are likely to be errors. These code patterns are known as “bug patterns”. Bug Patterns are checklist items that FindBugs uses to find problems in the Java code. There are many patterns defined in the tools which the user can select. In addition to the pre-defined rules, users are able to add their own rules to FindBugs and extend upon the already expansive set of rules. The bug patterns utilized by FindBugs can be categorized as follows:

- **Malicious code vulnerability:** Code that can be altered by other code maliciously.
- **Dodgy:** Code that can lead to unwanted errors.
- **Bad Practice:** Code that doesn’t adhere to accepted coding practices.
- **Correctness:** Code that produces different results than intended.
- **Internationalization:** Code that inhibits the se of international characters.
- **Performance:** Code that hinders performance.

- **Security:** Code that could result in security problems.
- **Multithreaded correctness:** Code that could cause problems in a multi-threaded environment.
- **Experimental:** Code that could possibly miss clean-up of streams, database objects, or other object that require cleanup operations.

The FindBugs plugin for eclipse offers a wide range of self-explanatory metrics to be configured so as to make the bug search more flexible towards the direction in which user is focused. Each metric has its speed of execution and description of the area it tests (Figure 1 and 2).



[✓] Enable project specific settings

[ ] Run automatically, [ ] (also on full build), analysis effort: Default

Store issue evaluations in: (cloud disabled)

Minimum rank to report: (1 is most severe, 20 is least) 15 (Of Concern)

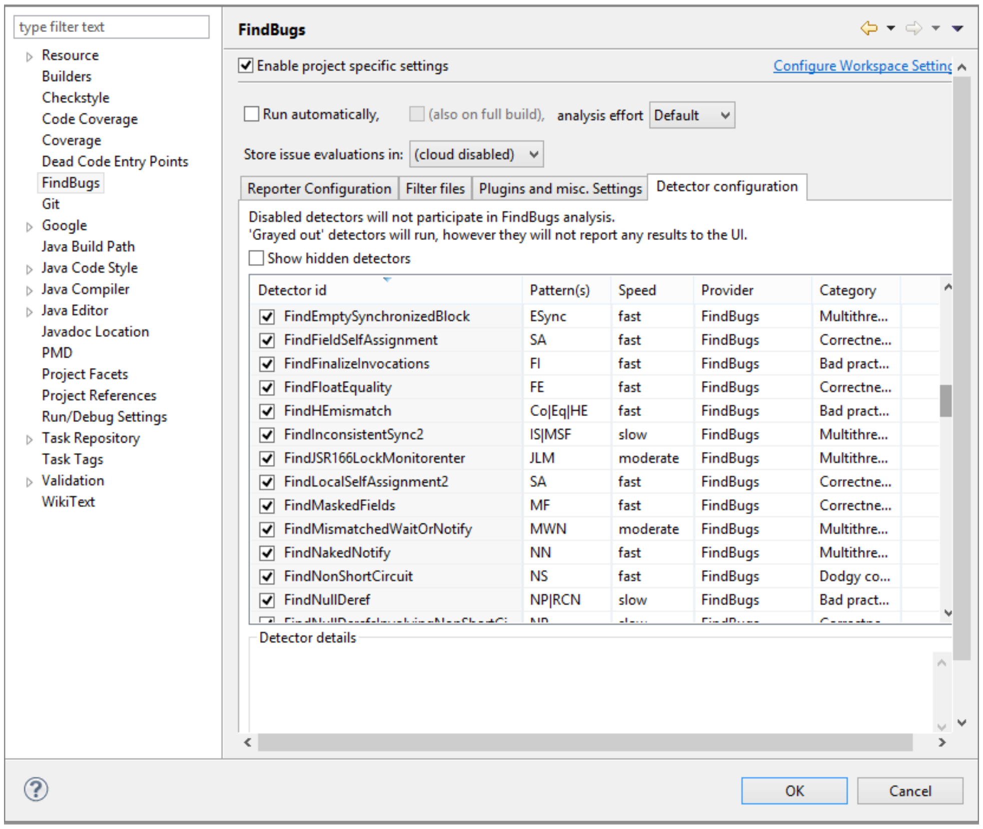
Minimum confidence to report: Medium

Reported (visible) bug categories
- [ ] Malicious code vulnerability
- [✓] Dodgy code
- [✓] Bad practice
- [✓] Correctness
- [ ] Internationalization
- [✓] Performance
- [ ] Security
- [✓] Multithreaded correctness
- [ ] Experimental

Mark bugs with ... rank as:

Scariest:	Warning
Scary:	Warning
Troubling:	Warning
Of concern:	Warning

Figure 1 FindBugs reporter configuration



FindBugs

☑ Enable project specific settings

☐ Run automatically, ☐ (also on full build), analysis effort Default

Store issue evaluations in: (cloud disabled)

Disabled detectors will not participate in FindBugs analysis.
'Grayed out' detectors will run, however they will not report any results to the UI.
☐ Show hidden detectors

Detector id	Pattern(s)	Speed	Provider	Category
☑ FindEmptySynchronizedBlock	ESync	fast	FindBugs	Multithre...
☑ FindFieldSelfAssignment	SA	fast	FindBugs	Correctne...
☑ FindFinalizeInvocations	FI	fast	FindBugs	Bad pract...
☑ FindFloatEquality	FE	fast	FindBugs	Correctne...
☑ FindHEmismatch	Co|Eq|HE	fast	FindBugs	Bad pract...
☑ FindInconsistentSync2	IS|MSF	slow	FindBugs	Multithre...
☑ FindJSR166LockMonitorenter	JLM	moderate	FindBugs	Multithre...
☑ FindLocalSelfAssignment2	SA	fast	FindBugs	Correctne...
☑ FindMaskedFields	MF	fast	FindBugs	Correctne...
☑ FindMismatchedWaitOrNotify	MWN	moderate	FindBugs	Multithre...
☑ FindNakedNotify	NN	fast	FindBugs	Multithre...
☑ FindNonShortCircuit	NS	fast	FindBugs	Dodgy co...
☑ FindNullDeref	NP|RCN	slow	FindBugs	Bad pract...

Figure 2 FindBugs configuration

CheckStyle

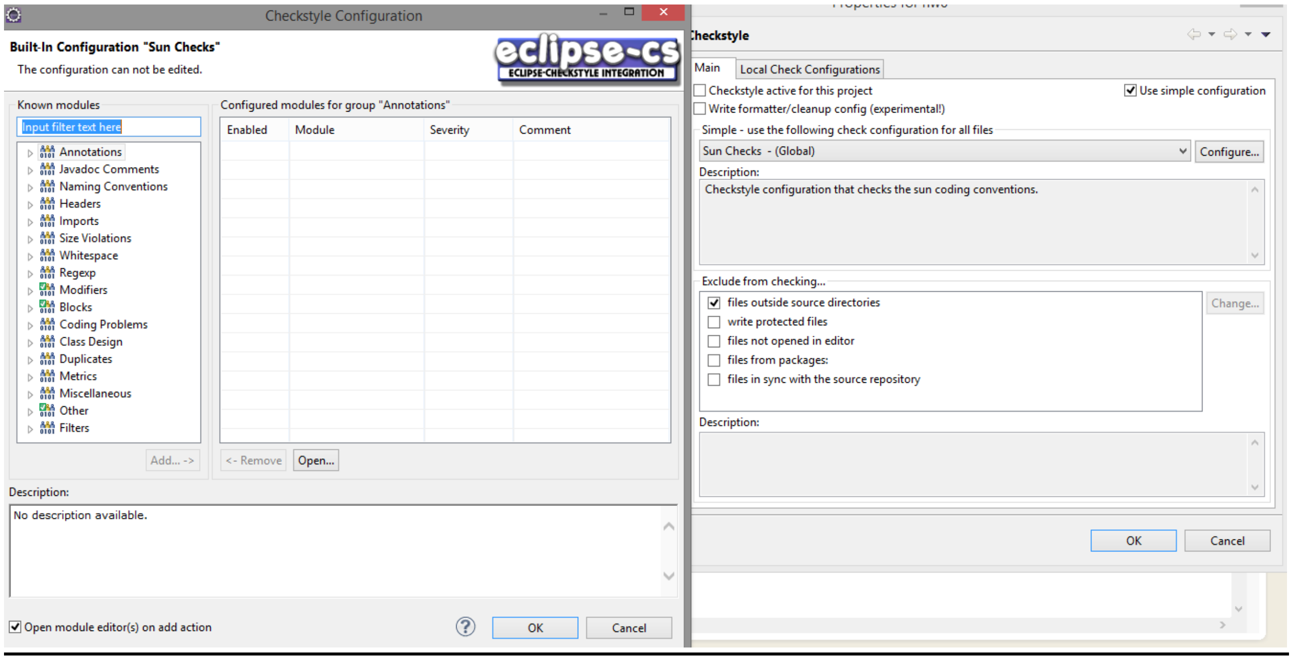
CheckStyle is a static code analyzer that helps assure that code adheres to a coding standard. It is written by Oliver Burn, Lars Kühne and is currently maintained by a team of developers around the world. It is written in JAVA and is under the Lesser GNU General Public License.

CheckStyle checks code by inspecting source code and pointing out items that deviate from a defined set of coding rules. Since CheckStyle is highly configurable and can be made to support any coding standard, it is ideal for users that want to enforce a certain coding standard for their project. Though CheckStyle is very useful in helping code maintain code quality, readability and reusability, the performed checks limit themselves to presentation related issues. These checks don’t analyze content, thus they don’t check for correctness or completeness of a code.

CheckStyle can check many different aspect of a source code. Its main functionalities are related to issues with code layout, design problems, duplicate code, and a variety of different bugs. CheckStyle provides many different checks that can be applied to any source bode. The following are examples of possible checks that CheckStyle provides.

- **AbstractClassName:** Ensures that the names of abstract classes conforming to some regular expression.
- **ClassTypeParameterName:** Checks that class type parameter names conform to a format specified by the format property.
- **DesignForExtension:** Check that classes are designed for inheritance.
- **FileLength:** Checks for long source files.
- **IllegalToken:** Checks for illegal tokens.
- **JUnitTestCase:** Ensures that the setUp(), tearDown()methods are named correctly, have no arguments, return void and are either public or protected.
- **MethodTypeParameterName:** Checks that class type parameter names conform to a format specified by the format property.
- **PackageDeclaration:** Ensures there is a package declaration and (optionally) in the correct directory.
- **UnusedImports:** Checks for unused import statements.

Since CheckStyle is more focused on the style conventions, it has its base rule-set as what is provided from Sun Microsystems as standards for JAVA which can also be configured and customized by the user.


Checkstyle Configuration

Built-In Configuration "Sun Checks"
The configuration can not be edited.

Known modules
Input filter text here
Annotations
Javadoc Comments
Naming Conventions
Headers
Imports
Size Violations
Whitespace
Regexp
Modifiers
Blocks
Coding Problems
Class Design
Duplicates
Metrics
Miscellaneous
Other
Filters

Configured modules for group "Annotations"

Enabled	Module	Severity	Comment

Description:
No description available.

☑ Open module editor(s) on add action

Checkstyle

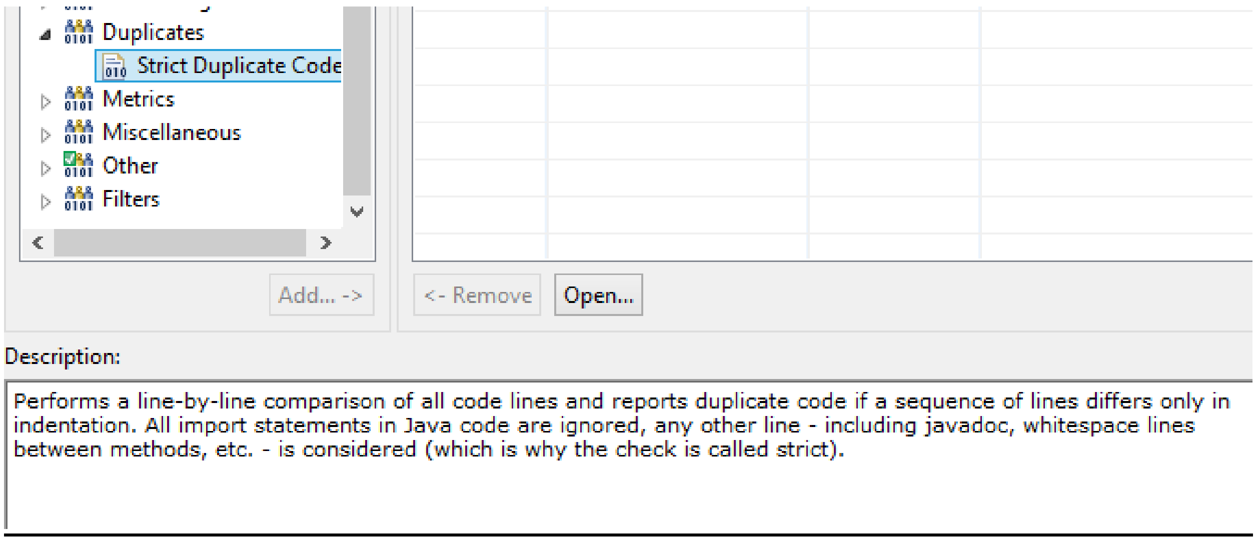
Main | Local Check Configurations

☐ Checkstyle active for this project
☐ Write formatter/cleanup config (experimental!)
☑ Use simple configuration

Simple - use the following check configuration for all files
Sun Checks - (Global)
Description:
Checkstyle configuration that checks the sun coding conventions.

Exclude from checking...
☑ files outside source directories
☐ write protected files
☐ files not opened in editor
☐ files from packages:
☐ files in sync with the source repository
Description:

Figure 3 CheckStyle Configuration



Performs a line-by-line comparison of all code lines and reports duplicate code if a sequence of lines differs only in indentation. All import statements in Java code are ignored, any other line - including javadoc, whitespace lines between methods, etc. - is considered (which is why the check is called strict).

Figure 4 CheckStyle example rule and description for code duplication

PMD

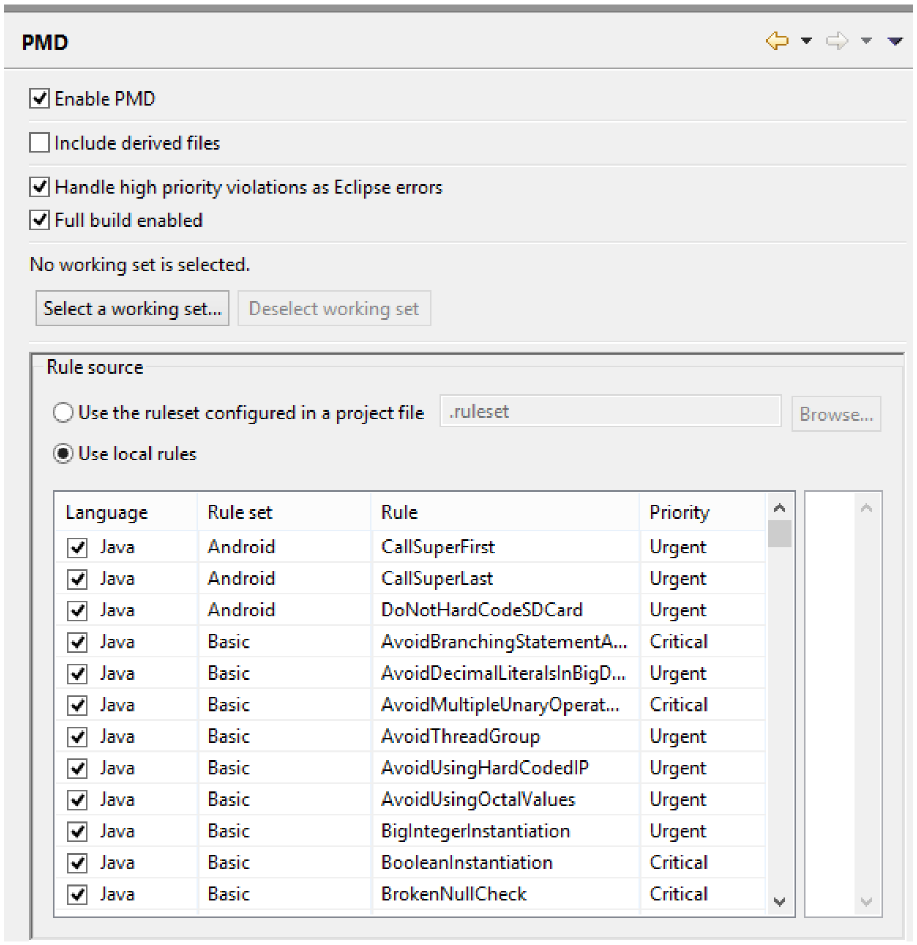
PMD is a static analyzer that looks at source code. It is based on the evaluation the code in accordance to a set of predetermined rules. It was initially written in support of Couggar, a Defense Advanced Research Projects Agency project, and eventually become a static code analyzer open to the public. PMD comes with a number of built in rules that can find a variety of code error. In addition, PMD also allows users to execute custom rules by allowing the development of new rules to be evaluated. It can be easily extensible by users by writing new bug detectors using Java/XPath.

PMD has plugins for JDeveloper, Eclipse, jEdit, JBuilder, Omnicore's CodeGuide, NetBeans/Sun Studio, IntelliJ IDEA, TextPad, Maven, Ant, Gel, JCreator, Hudson, Jenkins, SonarQube and Emacs.

Many of the errors PMD looks for are stylistic conventions that are suspicious and can potentially cause errors. PMD identifies potential problems such as:

- **Bugs:** Empty try, catch, finally, switch statements.
- **Dead Code:** Unused variables, parameters, and private methods.
- **Empty statements:** Empty if/while statements
- **Overcomplicated expressions:** Unnecessary if statements and for loops.
- **Suboptimal Code:** Wasteful strings/StringBuffer usage
- **Duplicate Code:** Copied/Pasted bugs

Unlike FindBugs and CheckStyle, PMD does not allow users to alter the rules predefined in the plugin but allows addition and removal of external rules. This means that the existing tools cannot be tweaked but can be removed and a variant of that rule with desired tweaks can be added to the rule set.



PMD

[✓] Enable PMD

[ ] Include derived files

[✓] Handle high priority violations as Eclipse errors

[✓] Full build enabled

No working set is selected.

Select a working set... | Deselect working set

Rule source

○ Use the ruleset configured in a project file .ruleset Browse...

● Use local rules

Language	Rule set	Rule	Priority
✓ Java	Android	CallSuperFirst	Urgent
✓ Java	Android	CallSuperLast	Urgent
✓ Java	Android	DoNotHardCodeSDCard	Urgent
✓ Java	Basic	AvoidBranchingStatementA...	Critical
✓ Java	Basic	AvoidDecimalLiteralsInBigD...	Urgent
✓ Java	Basic	AvoidMultipleUnaryOperat...	Critical
✓ Java	Basic	AvoidThreadGroup	Urgent
✓ Java	Basic	AvoidUsingHardCodedIP	Urgent
✓ Java	Basic	AvoidUsingOctalValues	Urgent
✓ Java	Basic	BigIntegerInstantiation	Urgent
✓ Java	Basic	BooleanInstantiation	Critical
✓ Java	Basic	BrokenNullCheck	Critical

Figure 5 PMD configuration and rule-set

<u>SonarGraph (Architect)</u>

SonarGraph is a static code analyzer that can be split into 2 variants, SonarGraph-Architect, which focuses on software structure and architecture, and SonarGraph-Quality, which adds additional metrics, quality, models, duplicate code search, integration of 3rd party code checkers, and other features. For the purpose of this report, we will focus on SonarGraph-Architect.

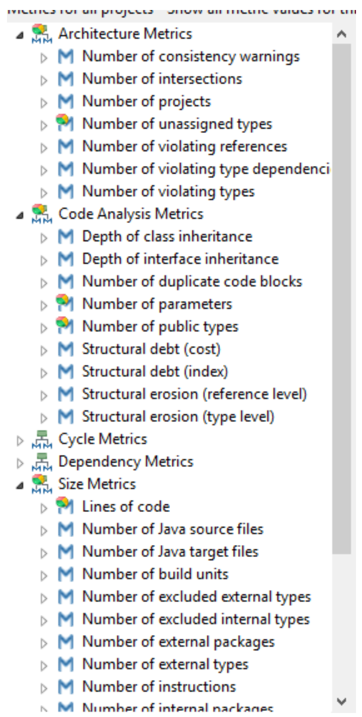
SonarGraph-Architect uses direct parsing to build as in-memory model of a software project. As a result of running everything in memory, SonarGraph is fast and efficient. It compares architecture with dependency structure of codes and lists all places where the code does no conform to the architecture. Developers are able to detect rule violations while working on code.

Using SonarGraph-architect will significantly improve the quality, maintainability, and comprehensibility of a user's code base. The coupling will be kept low and reuse of existing components will be greatly simplified. This will cut down the time needed to understand complex code and allow more time for implementation of useful functionality.

Typical use-cases for SonarGraph architect include:

- Monitoring of architecture rules and dependency structure.
- Assessment of structural quality of a code base.
- Analyze and visualize dependency structure of code and cyclic dependencies
- Break up cyclic dependencies with the last amount of effort
- Prepare and simulate complex refactoring to improve structure
- Define and enforce architectural rules and constraints for IDE during
- Monitor software metrics and use thresholds to create violations for bad values
- Integrate sophisticated architecture and metrics rule
- Find duplicate code

SonarGraph does not exist as an eclipse plugin alone but the plugin needs to be integrated with the standalone version of SonarGraph Architect for it to be useful. A subset of the plethora of customizable metrics provided in SonarGraph Architect is shown in Figure 6.



Metrics for all projects - Show all metric values for th

- Architecture Metrics
  - Number of consistency warnings
  - Number of intersections
  - Number of projects
  - Number of unassigned types
  - Number of violating references
  - Number of violating type dependenci
  - Number of violating types
- Code Analysis Metrics
  - Depth of class inheritance
  - Depth of interface inheritance
  - Number of duplicate code blocks
  - Number of parameters
  - Number of public types
  - Structural debt (cost)
  - Structural debt (index)
  - Structural erosion (reference level)
  - Structural erosion (type level)
- Cycle Metrics
- Dependency Metrics
- Size Metrics
  - Lines of code
  - Number of Java source files
  - Number of Java target files
  - Number of build units
  - Number of excluded external types
  - Number of excluded internal types
  - Number of external packages
  - Number of external types
  - Number of instructions
  - Number of internal packages

Figure 6 SonarGraph Architect metrics

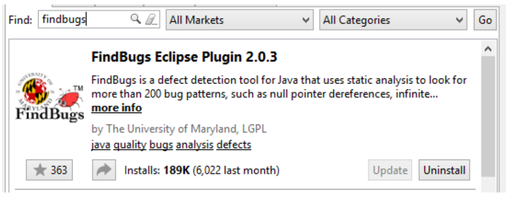
Section 2

Methods

In order to compare the 4 different static code analysis tools, we decided to test them using Eclipse as our IDE. We first downloaded the plugins from their respective sites and installed them in Eclipse. We wrote JAVA codes of varying lengths and content, tested for usability by using the JUnit tests for bugs and code coverage using EclEmma and then checked if the tools we chose could find the bugs, if any, in the code.

Installation and running options:

- *FindBugs*


Find: findbugs | All Markets | All Categories | Go

FindBugs Eclipse Plugin 2.0.3

FindBugs is a defect detection tool for Java that uses static analysis to look for more than 200 bug patterns, such as null pointer dereferences, infinite... **more info**

by The University of Maryland, LGPL

java quality bugs analysis defects

363 | Installs: **189K** (6,022 last month) | Update | Uninstall

Figure 7 FindBugs in Eclipse marketplace

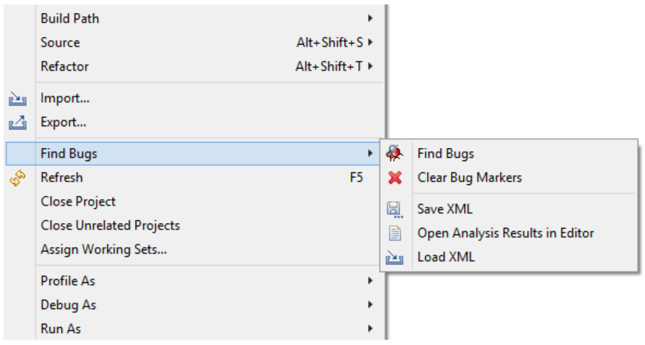
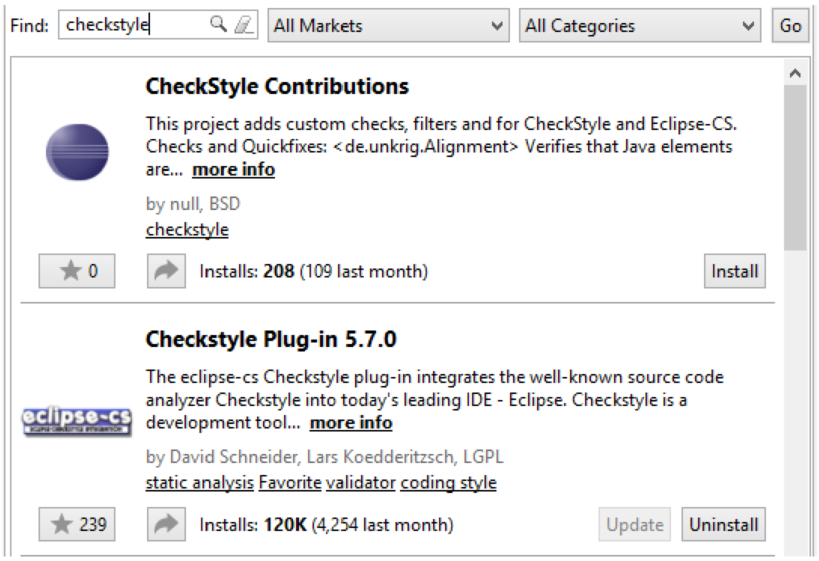


Figure 8 FindBugs options in Eclipse

- *CheckStyle:*


Find: checkstyle | All Markets | All Categories | Go

CheckStyle Contributions

This project adds custom checks, filters and for CheckStyle and Eclipse-CS. Checks and Quickfixes: <de.unkrig.Alignment> Verifies that Java elements are... **more info**

by null, BSD

checkstyle

0 | Installs: **208** (109 last month) | Install

Checkstyle Plug-in 5.7.0

The eclipse-cs Checkstyle plug-in integrates the well-known source code analyzer Checkstyle into today's leading IDE - Eclipse. Checkstyle is a development tool... **more info**

by David Schneider, Lars Koedderitzsch, LGPL

static analysis Favorite validator coding style

239 | Installs: **120K** (4,254 last month) | Update | Uninstall

Figure 99 CheckStyle in Eclipse Marketplace

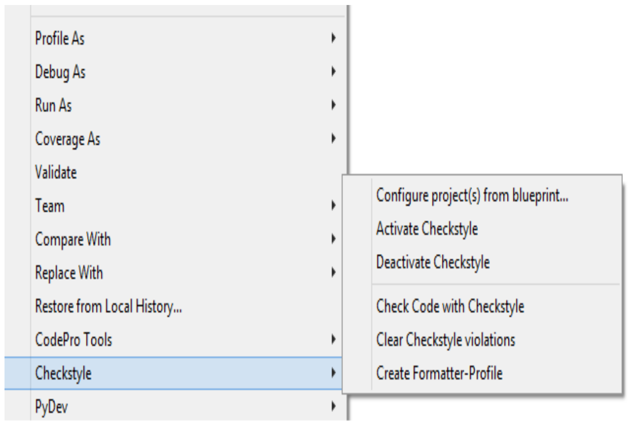
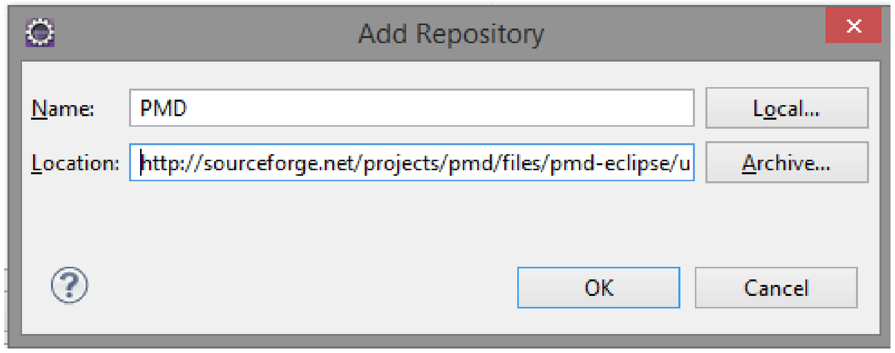


Figure 10 CheckStyle options

➢ *PMD*



Name: PMD

Location: http://sourceforge.net/projects/pmd/files/pmd-eclipse/u

Figure 11 PMD installation from 'Install new software' in Eclipse

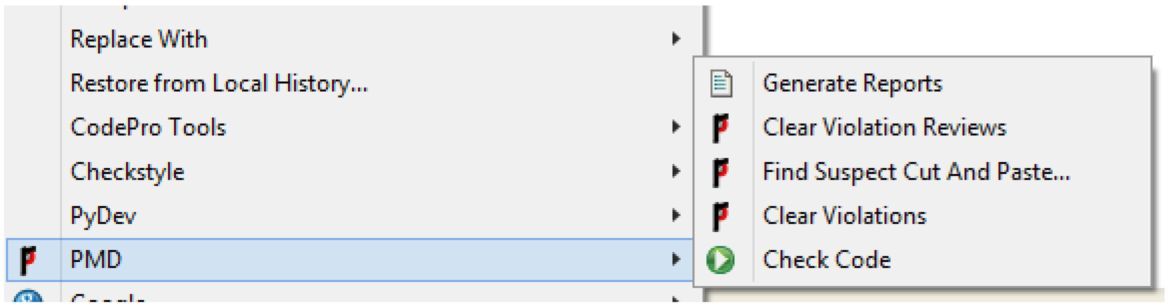


Figure 12 PMD options

- *SonarGraph:*

The software can be used free of cost on projects of smaller than 50000 bytecode instructions (with a community license) and is paid for analyzing projects of larger sizes. The software can be downloaded after registration from the hello2morrow website.

http://www.hello2morrow.com/products/sonargraph

The executable setup file is contained within the compressed zip archive that gets downloaded. Post-installation steps are as follows:

- Install the SonarGraph Eclipse plugin from SonarGraph Architect
- Create new system in SonarGraph Architect
- Set java root path to the concerned eclipse project and save the system as .sonargraph file
- Go to eclipse and link the project there too using the system file created in last step.
- Start using SonarGraph in Eclipse on the project.

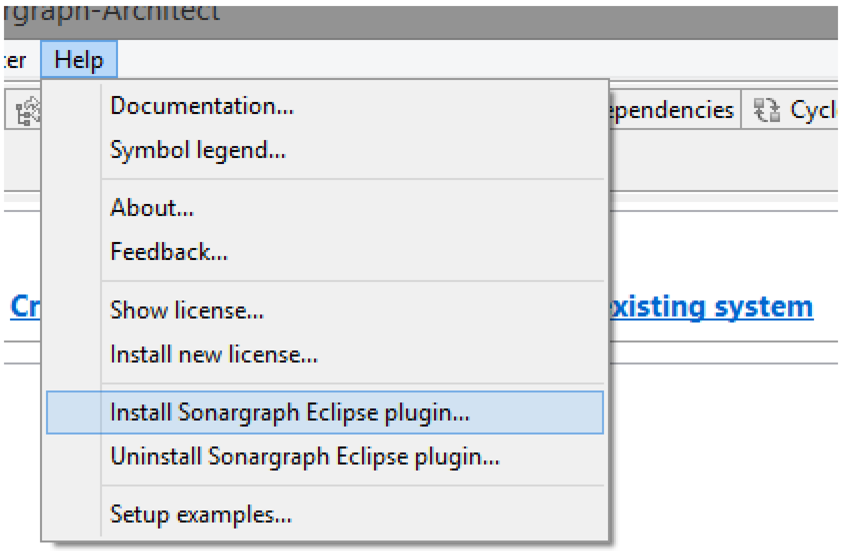
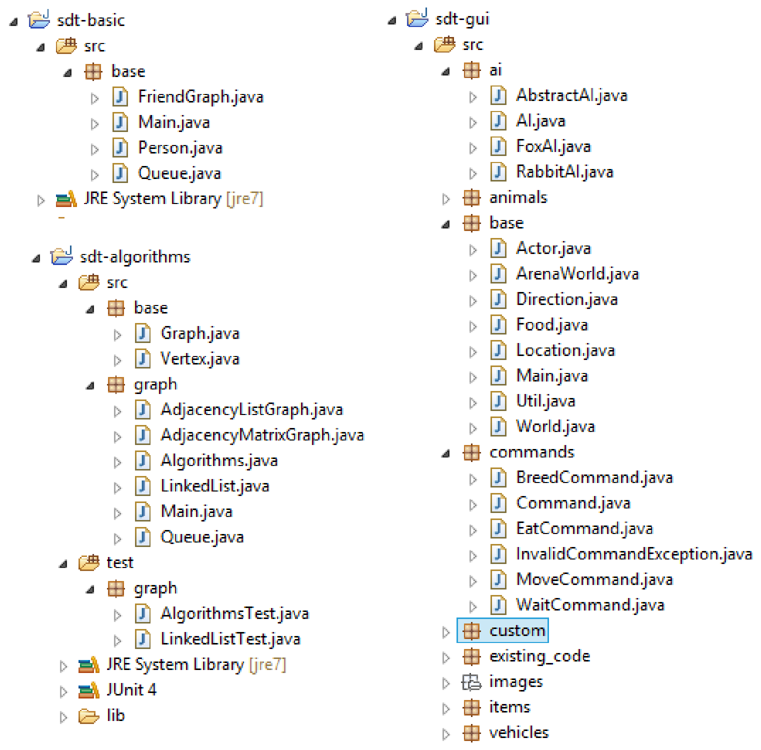


Figure 10 SonarGraph eclipse plugin installation

Code description: The code structure for the three categories of code sizes is given as follows:



```
sdt-basic
  src
    base
      FriendGraph.java
      Main.java
      Person.java
      Queue.java
  JRE System Library [jre7]

sdt-algorithms
  src
    base
      Graph.java
      Vertex.java
    graph
      AdjacencyListGraph.java
      AdjacencyMatrixGraph.java
      Algorithms.java
      LinkedList.java
      Main.java
      Queue.java
  test
    graph
      AlgorithmsTest.java
      LinkedListTest.java
  JRE System Library [jre7]
  JUnit 4
  lib

sdt-gui
  src
    ai
      AbstractAI.java
      AI.java
      FoxAI.java
      RabbitAI.java
    animals
    base
      Actor.java
      ArenaWorld.java
      Direction.java
      Food.java
      Location.java
      Main.java
      Util.java
      World.java
    commands
      BreedCommand.java
      Command.java
      EatCommand.java
      InvalidCommandException.java
      MoveCommand.java
      WaitCommand.java
    custom
    existing_code
    images
    items
    vehicles
```

Figure 11 Code description

The total lines of code in sdt-basic were 136, in sdt-algorithms were 783 and for sdt-gui were 1835. While we tested the basic sources of bugs in the small sized code, we tested bugs that are not very subtle in large codes.

The snapshots shown here are mainly for the sdt-gui project since it was the biggest out of all of them with most potential for the more obscure bugs. The running version of the project is shown as:

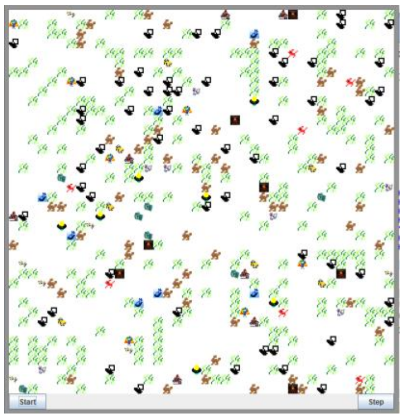
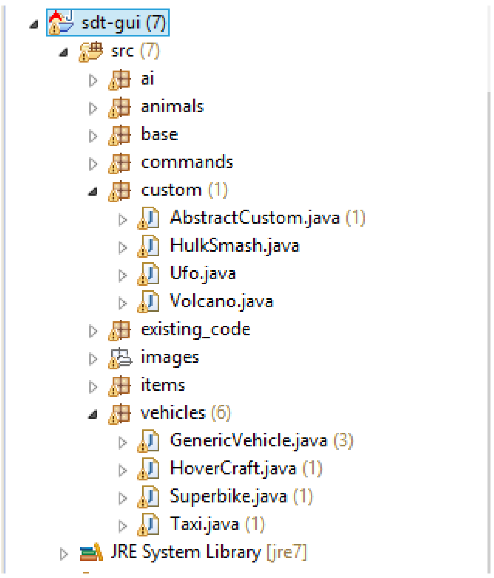


Figure 12 sdt-gui running snapshot

Section 3

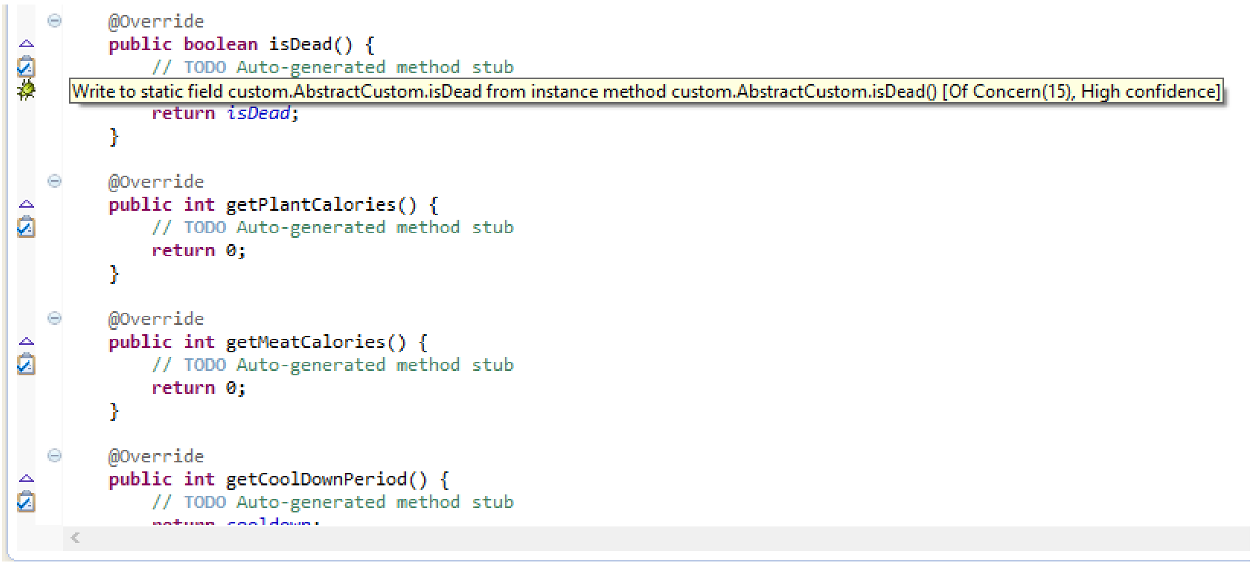
Results

FindBugs



```
sdt-gui (7)
  src (7)
    ai
    animals
    base
    commands
    custom (1)
      AbstractCustom.java (1)
      HulkSmash.java
      Ufo.java
      Volcano.java
    existing_code
    images
    items
    vehicles (6)
      GenericVehicle.java (3)
      HoverCraft.java (1)
      Superbike.java (1)
      Taxi.java (1)
  JRE System Library [jre7]
```

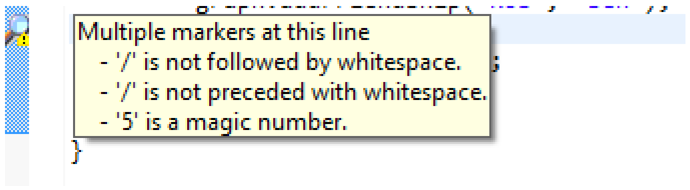
Figure 16 Project view after running FindBugs. The numbers represent the number of errors in the hierarchy.



```
    @Override
    public boolean isDead() {
        // TODO Auto-generated method stub
Write to static field custom.AbstractCustom.isDead from instance method custom.AbstractCustom.isDead() [Of Concern(15), High confidence]
        return isDead;
    }

    @Override
    public int getPlantCalories() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public int getMeatCalories() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public int getCoolDownPeriod() {
        // TODO Auto-generated method stub
```

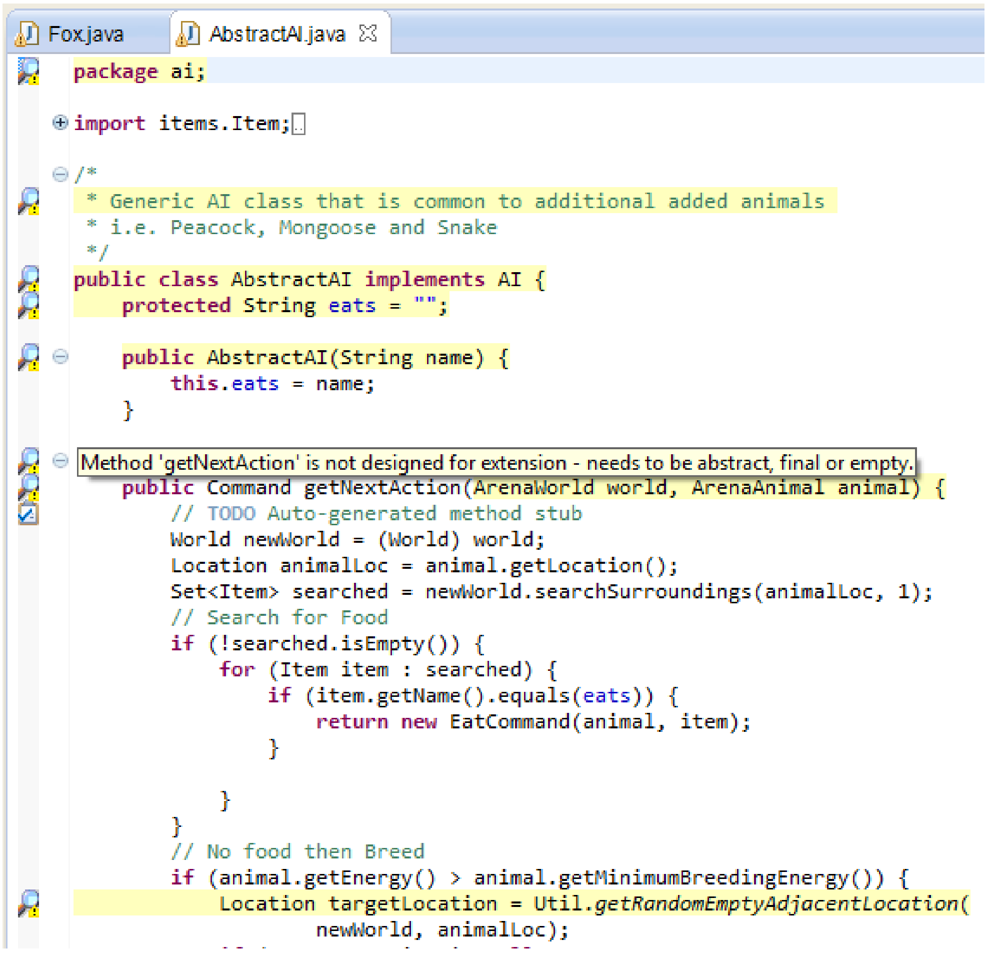
Figure 17 Bug from FindBugs. Serious bug.

CheckStyle



```
Multiple markers at this line
  - '/' is not followed by whitespace.
  - '/' is not preceded with whitespace.
  - '5' is a magic number.
}
```

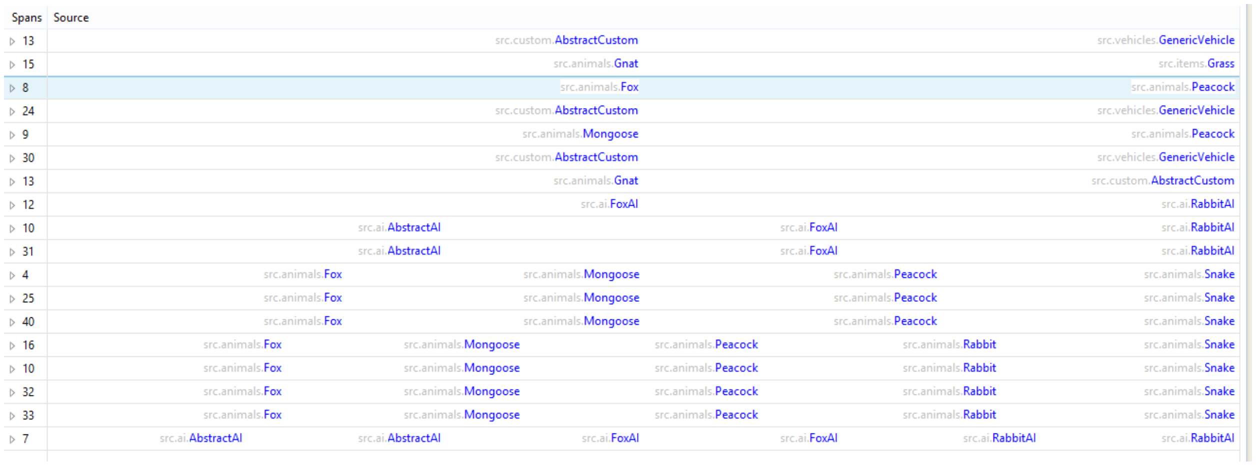
Figure 18 On a line 'int a = 5/0;'



```
package ai;

import items.Item;

/*
 * Generic AI class that is common to additional added animals
 * i.e. Peacock, Mongoose and Snake
 */
public class AbstractAI implements AI {
    protected String eats = "";

    public AbstractAI(String name) {
        this.eats = name;
    }

Method 'getNextAction' is not designed for extension - needs to be abstract, final or empty.
    public Command getNextAction(ArenaWorld world, ArenaAnimal animal) {
        // TODO Auto-generated method stub
        World newWorld = (World) world;
        Location animalLoc = animal.getLocation();
        Set<Item> searched = newWorld.searchSurroundings(animalLoc, 1);
        // Search for Food
        if (!searched.isEmpty()) {
            for (Item item : searched) {
                if (item.getName().equals(eats)) {
                    return new EatCommand(animal, item);
                }

            }
        }
        // No food then Breed
        if (animal.getEnergy() > animal.getMinimumBreedingEnergy()) {
            Location targetLocation = Util.getRandomEmptyAdjacentLocation(
                    newWorld, animalLoc);
```

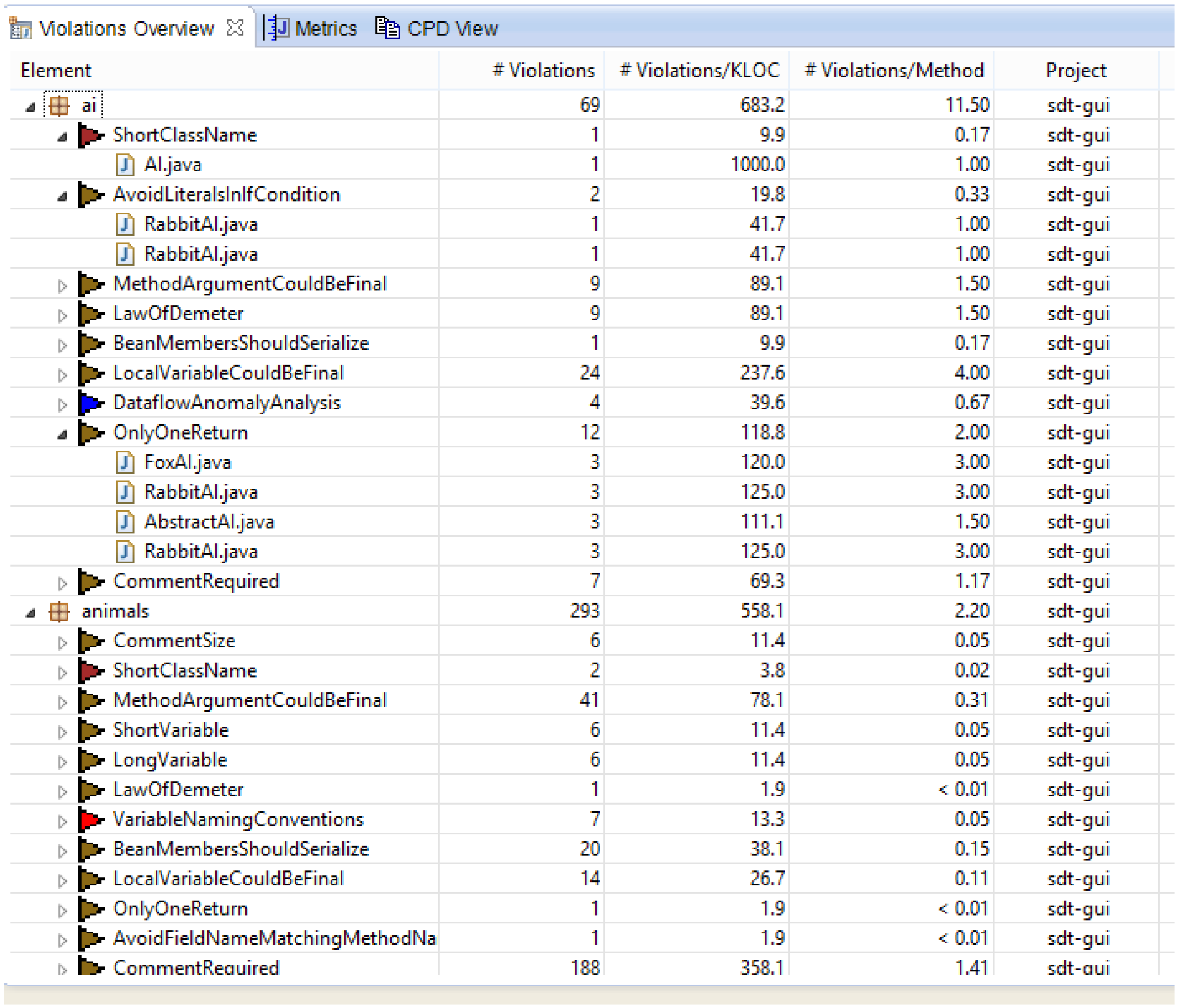
Figure 19 Sample error with CheckStyle Not an actual bug but still reported as a bug.

PMD



Spans	Source					
13	src.custom.AbstractCustom	src.vehicles.GenericVehicle				
15	src.animals.Gnat	src.items.Grass				
8	src.animals.Fox	src.animals.Peacock				
24	src.custom.AbstractCustom	src.vehicles.GenericVehicle				
9	src.animals.Mongoose	src.animals.Peacock				
30	src.custom.AbstractCustom	src.vehicles.GenericVehicle				
13	src.animals.Gnat	src.custom.AbstractCustom				
12	src.ai.FoxAI	src.ai.RabbitAI				
10	src.ai.AbstractAI	src.ai.FoxAI	src.ai.RabbitAI			
31	src.ai.AbstractAI	src.ai.FoxAI	src.ai.RabbitAI			
4	src.animals.Fox	src.animals.Mongoose	src.animals.Peacock	src.animals.Snake		
25	src.animals.Fox	src.animals.Mongoose	src.animals.Peacock	src.animals.Snake		
40	src.animals.Fox	src.animals.Mongoose	src.animals.Peacock	src.animals.Snake		
16	src.animals.Fox	src.animals.Mongoose	src.animals.Peacock	src.animals.Rabbit	src.animals.Snake	
10	src.animals.Fox	src.animals.Mongoose	src.animals.Peacock	src.animals.Rabbit	src.animals.Snake	
32	src.animals.Fox	src.animals.Mongoose	src.animals.Peacock	src.animals.Rabbit	src.animals.Snake	
33	src.animals.Fox	src.animals.Mongoose	src.animals.Peacock	src.animals.Rabbit	src.animals.Snake	
7	src.ai.AbstractAI	src.ai.AbstractAI	src.ai.FoxAI	src.ai.FoxAI	src.ai.RabbitAI	src.ai.RabbitAI

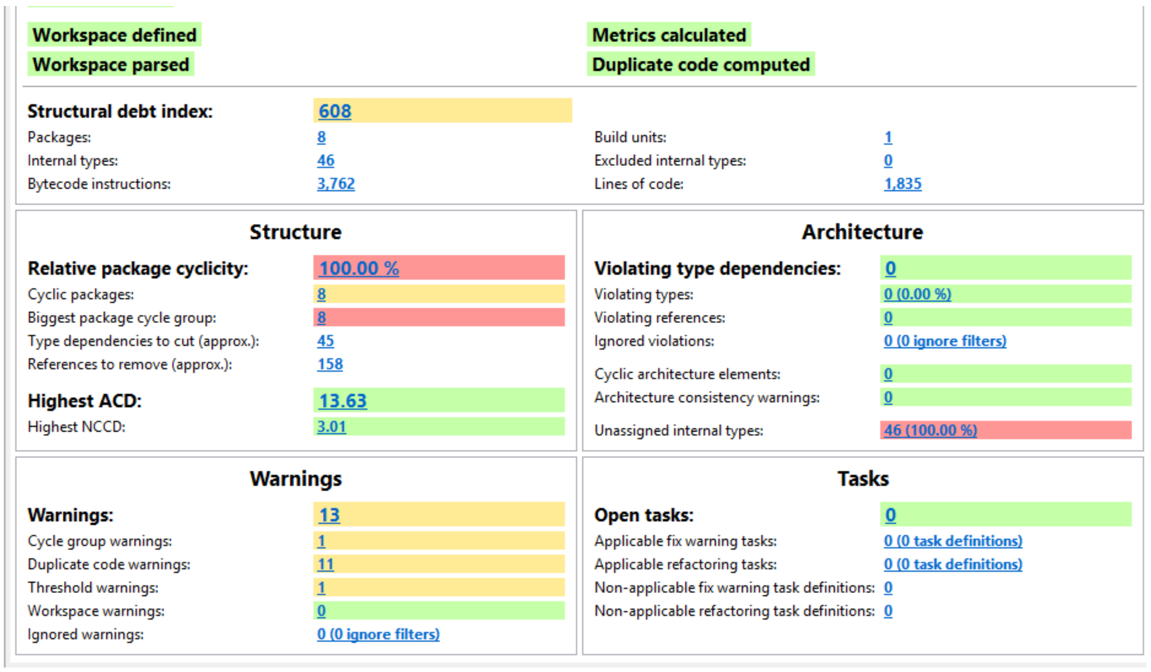
Figure 13 PMD code duplication report



Element	# Violations	# Violations/KLOC	# Violations/Method	Project
ai	69	683.2	11.50	sdt-gui
ShortClassName	1	9.9	0.17	sdt-gui
AI.java	1	1000.0	1.00	sdt-gui
AvoidLiteralsInIfCondition	2	19.8	0.33	sdt-gui
RabbitAI.java	1	41.7	1.00	sdt-gui
RabbitAI.java	1	41.7	1.00	sdt-gui
MethodArgumentCouldBeFinal	9	89.1	1.50	sdt-gui
LawOfDemeter	9	89.1	1.50	sdt-gui
BeanMembersShouldSerialize	1	9.9	0.17	sdt-gui
LocalVariableCouldBeFinal	24	237.6	4.00	sdt-gui
DataflowAnomalyAnalysis	4	39.6	0.67	sdt-gui
OnlyOneReturn	12	118.8	2.00	sdt-gui
FoxAI.java	3	120.0	3.00	sdt-gui
RabbitAI.java	3	125.0	3.00	sdt-gui
AbstractAI.java	3	111.1	1.50	sdt-gui
RabbitAI.java	3	125.0	3.00	sdt-gui
CommentRequired	7	69.3	1.17	sdt-gui
animals	293	558.1	2.20	sdt-gui
CommentSize	6	11.4	0.05	sdt-gui
ShortClassName	2	3.8	0.02	sdt-gui
MethodArgumentCouldBeFinal	41	78.1	0.31	sdt-gui
ShortVariable	6	11.4	0.05	sdt-gui
LongVariable	6	11.4	0.05	sdt-gui
LawOfDemeter	1	1.9	< 0.01	sdt-gui
VariableNamingConventions	7	13.3	0.05	sdt-gui
BeanMembersShouldSerialize	20	38.1	0.15	sdt-gui
LocalVariableCouldBeFinal	14	26.7	0.11	sdt-gui
OnlyOneReturn	1	1.9	< 0.01	sdt-gui
AvoidFieldNameMatchingMethodNa	1	1.9	< 0.01	sdt-gui
CommentRequired	188	358.1	1.41	sdt-gui

Figure 14 PMD violations report

SonarGraph



Workspace defined
Workspace parsed
Metrics calculated
Duplicate code computed

Structural debt index:	608		
Packages:	8	Build units:	1
Internal types:	46	Excluded internal types:	0
Bytecode instructions:	3,762	Lines of code:	1,835

Structure

Relative package cyclicity:	100.00 %
Cyclic packages:	8
Biggest package cycle group:	8
Type dependencies to cut (approx.):	45
References to remove (approx.):	158
Highest ACD:	13.63
Highest NCCD:	3.01

Architecture

Violating type dependencies:	0
Violating types:	0 (0.00 %)
Violating references:	0
Ignored violations:	0 (0 ignore filters)
Cyclic architecture elements:	0
Architecture consistency warnings:	0
Unassigned internal types:	46 (100.00 %)

Warnings

Warnings:	13
Cycle group warnings:	1
Duplicate code warnings:	11
Threshold warnings:	1
Workspace warnings:	0
Ignored warnings:	0 (0 ignore filters)

Tasks

Open tasks:	0
Applicable fix warning tasks:	0 (0 task definitions)
Applicable refactoring tasks:	0 (0 task definitions)
Non-applicable fix warning task definitions:	0
Non-applicable refactoring task definitions:	0

Figure 15 Sonargraph-Architect dashboard after parsing the project

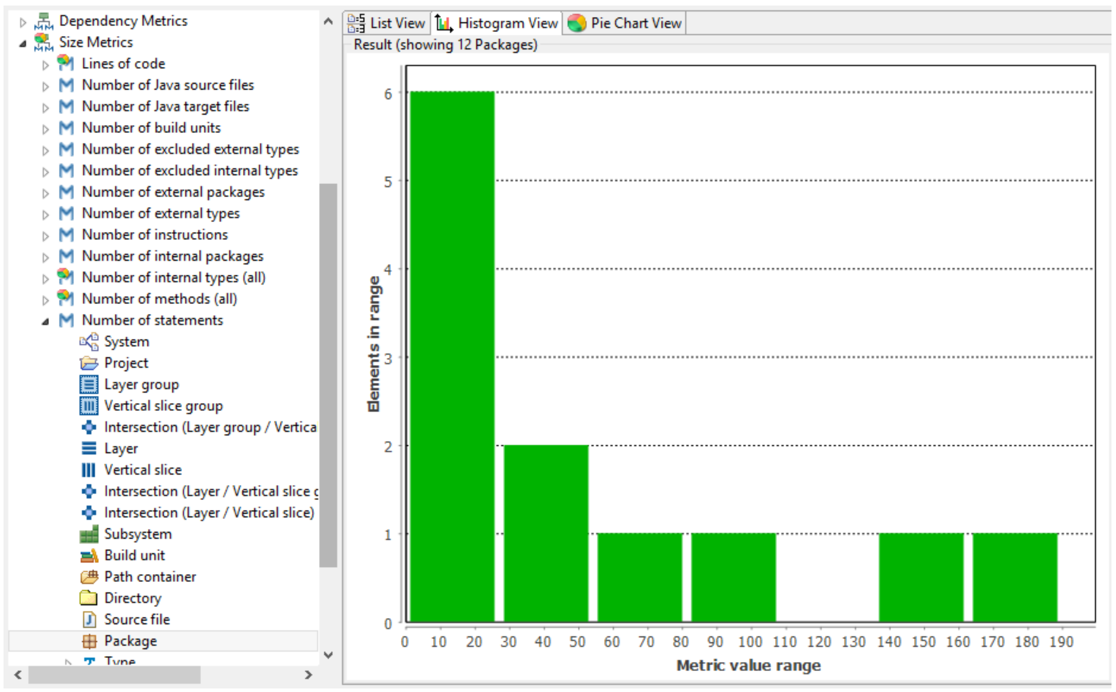
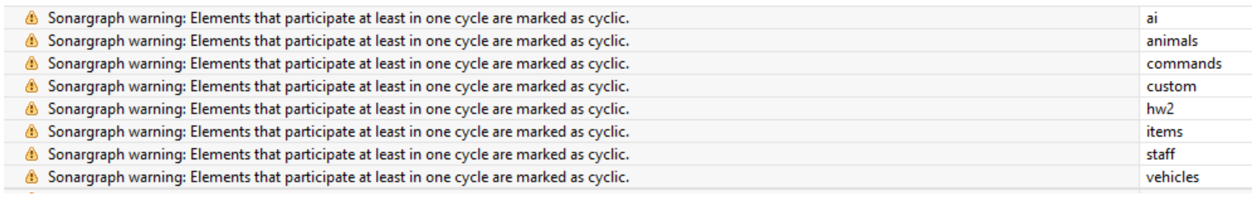
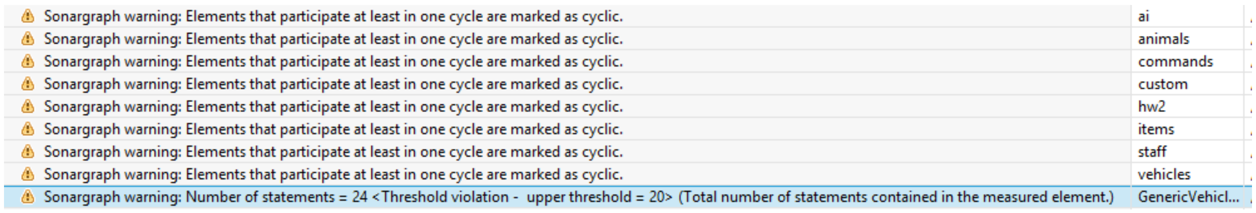


Figure 16 Sonargraph metrics subset and graphical report



Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	ai
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	animals
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	commands
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	custom
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	hw2
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	items
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	staff
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	vehicles

Figure 17 Sonargraph default metrics warnings



Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	ai
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	animals
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	commands
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	custom
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	hw2
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	items
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	staff
Sonargraph warning: Elements that participate at least in one cycle are marked as cyclic.	vehicles
Sonargraph warning: Number of statements = 24 <Threshold violation - upper threshold = 20> (Total number of statements contained in the measured element.)	GenericVehicl...

Figure 18 Sonargraph warnings with method length threshold = 20.

Comparison

	FindBugs	**CheckStyle**	**PMD**	**SonarGraph**
Null pointer dereferences	Yes	No	Yes	Yes
Class/Method/Variable nature	No	Yes	Yes	No
Duplicated code	No	Kind of	Yes	Yes
Blank lines and whitespace	No	Yes	No	No
Data Flow	Only recently	No	No	Yes
Optimization possibility	Yes	No	Yes	Yes
Number of Rules	414	132	234	>500
Requirement	Compiled Code	Uncompiled code	Uncompiled code	Compiled Code
Ease of Use (1-10)	9	7	8	4
Loops, indices, reachability	Yes	No	Yes	Yes
Extra Return statement	No	No	Yes	No
Naming Conventions	No	Yes	Yes	No

Table 1 Comparison of the tests done on the software

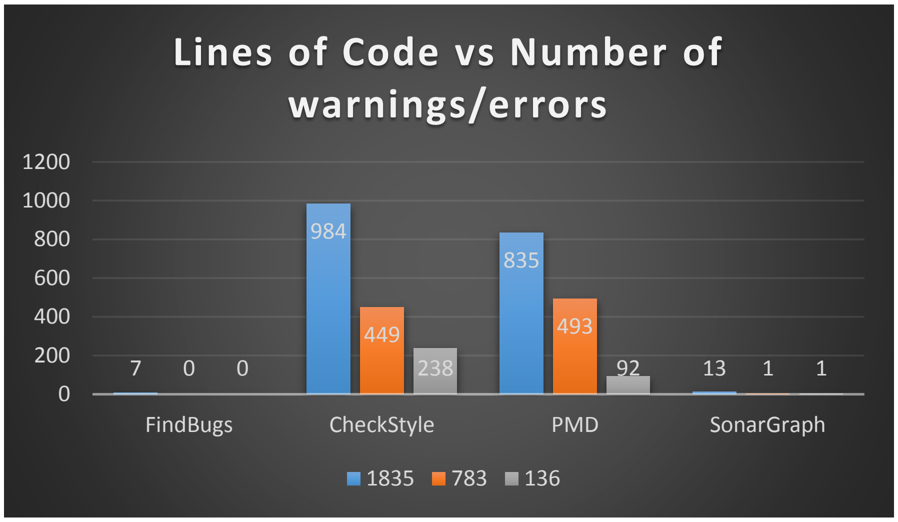


Figure 19: Performance comparison of Tools against code length

<u>**Conclusion**</u>

From the test results and comparisons, it is evident that for making sure that the project under consideration is working and bug-free, the best tool to use is FindBugs for small team projects. Although FindBugs reports many more warnings, the software itself considers them as low priority, hides them normally, and produces them only after in-depth tweaking of configurations. That said, it still does not produce results in a very easy to read format.

SonarGraph, also being a flow based analysis tool, produces warnings on cyclic dependencies etc. Even if we consider the areas which are commonly covered by all the tools, the ranking does not change. This is a very important conclusion since the only major change considering these areas across all 4 tools is the number of rules defined in the software itself. This is an indicative of a highly probable direct linear relationship between the number of rules defined in the software and the quality of the bugs reported with respect to meaningful execution.

Having considered the code execution aspect, which is indeed very important, we come to maintenance of the code. Inadequate documentation and non-compliance with the style conventions defeats the purpose of having coding standards when it comes to producing code that can be communicated and developed by other developers. Producing good quality code according to the standards is quintessential for team based works. Thus, it would be incorrect to say that a particular tool (SonarGraph/FindBugs) is better over the other (CheckStyle and PMD).

While all the tools have overlapping areas of operation and code analysis, each one has a specific feature that distinguishes from the rest, be it better prediction of style to finding data flow dependencies.

Future Directions

While we tried to be exhaustive in our tests without depending on external sources, given the timeframe, we were not able to test multithreaded code and massive open source java projects like apache/Netbeans etc. Keeping in mind the conclusions from this project regarding the linear relationship between rules and quality, future work should be directed towards more complex applications and verifying if the results from those tally with those produced here.

References

1. http://findbugs.sourceforge.net/
2. http://checkstyle.sourceforge.net/
3. http://pmd.sourceforge.net/
4. http://www.hello2morrow.com/products/sonarj
5. http://www.eclipse.org/