# Google Summer of Code, 2022

# Proposal for Checkstyle

## Project: Practice What You Preach

By:

Vyom Yadav

GitHub: Vyom-Yadav

LinkedIn: Vyom Yadav

Time Zone: Indian Standard Time (IST), UTC +5:30

## ABOUT ME

I’m Vyom Yadav, a second-year student pursuing a bachelor's in engineering (B.E.) in Computer Science from Chandigarh University, India.

I am an open-source enthusiast who loves to explore different projects and different domains. Checkstyle in particular caught my attention as the organization is beginner friendly and is well documented. Also, the project emphasizes the importance of clean code which I caught up in the early stage of my journey of writing code.

I am currently learning to build Microservices with the help of the Spring framework. I am also learning to use docker and Kubernetes along with Microsoft Azure.

## PROJECT DETAILS

The same can be found on Checkstyle's wiki page.

There are a lot of static analysis tools for Java language, it is not a problem to activate them in the project and get a report on what to improve. Problem is the next step, how to start fixing reported problems gradually and at the same time keep focusing on feature delivery as before and do not let anybody else contribute new violations while the team is fixing old problems.

Finding time to fix old problems is a hard but possible task. Not all engineers agree to spend time on this so usually small part of engineers start doing this to let others see the benefits of this and later the amount of involved engineers is growing. The most frustrating for engineers fact is that while resolving such violations some other contributors create violations (most of the time unintentionally). Fixing problems after code is merged to a common code base is a few times harder when doing it at the time of initial implementation.

The goal of this project is to find ways to activate more analysis tools in our code and find a way to enforce more strict rules step by step to not interfere with other engineers, and not let other engineers contribute new violations if some rules are enforced in a certain part of code.

Exact tasks with the goal above in mind:

1. Move Teamcity inspections config be based on a configuration file in a code, explain all suppression for inspections in separate Javadoc tag to be explicit why a violation is false-positive or won't fix.
2. Enforce 0 violations by Error-Prone over our code.
3. Use of ArchUnit in UT for design verification over classes and between classes.
4. Extend the usage of pitest to activate all mutators and cover all survivals by tests, and resolve all existing suppressions.
5. Activate the Checker Framework and fix violations from it.

## PROJECT TIMELINE

| Period | Proposed Work |
|---|---|
| May 20 – June 12 | This is the community bonding period.<br><br>I will get to reading documentation of Teamcity inspections, errorprone, archunit, checker framework, and pitest so we can move smoothly later on.<br>The order of completing tasks will be<br>1. Teamcity inspections<br>2. pitest<br>3. Error-Prone<br>4. ArchUnit<br>5. Checker Framework |
| June 13 – June 20 | Move team city inspections to a configuration file and suppress the violations which are false positives with the help of Javadoc tags.<br><br>A list of problem-causing inspections will be created and the fix will be sent one by one. |
| June 21 – July 25 | Extend usage of pitest to activate all mutators and cover all survival by tests, remove existing suppressions. The detailed process of extending pitest to all active mutators is mentioned here.<br><br>● Week 1 & 2: Remove[1] existing suppressions, several issues are open regarding this in the codebase. #6294, #6295, #6320, #6658, etc. |

[1] Removing all the existing suppressions won't be possible in the initial period. Indentation handlers are tightly coupled and already have many unsolved issues in the issue tracker. An attempt to remove suppressions for such modules would be done in a special phase at the end of the project.

| | |
|---|---|
| | ● After week 2: Extend pitest to use all stable mutators, and fix violations from those. Add suppressions for violations that are for the lines that add some conditions to reduce time complexity or for lines that fall under the wont-fix[2] category. |
| July 26-<br>July 29 | Phase 1 evaluation period. Mentors and GSoC contributors can begin submitting their evaluations of phase 1.<br><br>Tasks completed-<br><br>1. Move team city inspections to a configuration file.<br>2. Extend pitest to use all active mutators. |
| August 1 –<br>August 14 | Set up Error-Prone, and enforce it over the code-base. Fix violations on a module after module basis. Add suppressions that are a false positive. Detailed analysis of introducing error-prone is here. |
| August 15-<br>Sept 7 | Add ArchUnit testing for verification of design over classes and between classes.<br><br>ArchUnit will be used in combination with JUnit 4 and JUnit 5 as ArchUnit provides extended support for writing tests with JUnit 4 and JUnit 5.<br><br>Examples of ArchUnit tests are mentioned here. |
| Sep 8-<br>Oct 7 | Activate Checker Framework. The detailed process of activating it is mentioned here. |
| Oct 8 –<br>Nov 13 | **Special Phase-**<br><br>All the suppressions that were transferred earlier will be resolved in this phase. Some violations from the Checker framework and Error-Prone may also be resolved in this period. |

[2] This category will be investigated again in the special phase at the end of the project.

| | |
|---|---|
| | There is a high probability that these violations will be from Indentation modules, Javadoc checks, import checks, etc.<br><br>Rewriting these modules from scratch to make use of a better design pattern that is easier to maintain is in itself a GSoC project.<br><br>Partially rewriting these modules to remove redundant logical conditions, improve logical conditions, removing the usage of extensive boolean flags will be done in this phase to remove suppressions. |
| Nov 13 – Nov 21 | The project will be wrapped up and final reports will be submitted. |

## EXTENDING PITEST TO ALL ACTIVE MUTATORS

Pitest currently has the following mutators-

- Conditionals Boundary - Enabled in all profiles
- Increments - Enabled in all profiles
- Invert Negatives - Enabled in all profiles
- Math - Enabled in all profiles
- Negate Conditionals - Enabled in all profiles
- Return Values - Enabled in all profiles
- Void Method Calls - Enabled in all profiles
- Empty returns - Not activated, will be activated
- False Returns - Enabled in all profiles
- True returns - Enabled in all profiles
- Null returns - Not activated, will be activated
- Primitive returns - Not activated, will be activated
- Remove Conditionals - Enabled in all profiles except, pitest-indentation, pitest-javadoc, pitest-utils, #6320 focuses on introducing this mutator.
- Experimental Switch - Experimental mutators won't be added.
- Inline Constant - Not activated, will be activated

- Constructor Calls - Fairly unstable and likely to cause **NullPointerExceptions** even with weak test suites. Won't be added.
- Non Void Method Calls - Fairly unstable for some types (especially Objects where **NullPointerExceptions** are likely) and may also create equivalent mutations if it replaces a method that already returns one of the default values without also having a side effect. Won't be added.
- Remove Increments - Not activated, will be activated
- Experimental Argument Propagation - Experimental mutators won't be added.
- Experimental Big Integer - Experimental mutators won't be added.
- Experimental Member Variable - Experimental mutators won't be added.
- Experimental Naked Receiver - Experimental mutators won't be added.
- Negation - Not activated, will be activated
- Arithmetic Operator Replacement - Not activated, will be activated
- Arithmetic Operator Deletion - Not activated, will be activated
- Constant Replacement - Not activated, will be activated
- Bitwise Operator - Not activated, will be activated
- Relational Operator Replacement - Not activated, will be activated
- Unary Operator Insertion - Not activated, will be activated

Apart from these mutators, there are other mutators which are offered by arcmutate. Arcmutate extends and improves the open-source pitest framework, making the world's leading mutation testing tool for java even better. It introduces new operators that target the Streams API, predicate logic, and chained method calls, creating mutations in code that standard pitest cannot.

pitest takes a lot of time to execute, and arcmutate offers faster analysis with the CDG Accelerator Plugin. It claims to reduce the time of analysis for Commons Lang from over 55 minutes to under 18 minutes.

arcmutate is **free** for open source projects but a license needs to be acquired, if we agree to add arcmutate, this task (acquiring a license) would be done by the mentors. More about licenses and subscription plans can be found at arcmutate-pricing.

arcmutate offers the following mutators-

- CHAINED_CALLS
- REMOVE_DISTINCT
- REMOVE_FILTER
- REMOVE_LIMIT
- REMOVE_SKIP
- REMOVE_SORTED
- REMOVE_PREDICATE_NEGATION
- REMOVE_PREDICATE_AND
- REMOVE_PREDICATE_OR
- FIELD_WRITES
- SWAP_PARAMS
- SWAP_ANY_MATCH
- SWAP_PREDICATE_OR
- SWAP_PREDICATE_AND

These mutators will be added only after utilizing all the standard pitest mutators. The CDG Accelerator Plugin will be introduced at the beginning of adding new mutators, it will help reduce the execution time.

## IMPROVING THE SUPPRESSION LIST

At present I have no idea for improving the current suppression list, pitest out of the box does not support the suppression list, if we are able to add arcmutate mutators, a suppression list for them will also be created in a similar manner.

## STRATEGY FOR ADDING NEW MUTATORS

Separate issue for every new mutator to be added will be created. All the surviving mutations will be killed/suppressed in small PRs.

The standard approach to kill mutations will be to generate diff report with that mutation present in the code (eg - replacing condition with true). If a difference is found, those examples would be added to unit testing. If not then the logic will be investigated to kill the mutation.

If the mutation is very simple then diff report won’t be created as it consumes a lot of time and resources.

## ACTIVATING ERROR-PRONE

Error-prone will be used alongside the maven-compiler plugin in the default build. It comes with default bug patterns (what we call checks).

Only a single error was thrown after the enabling error-prone. In general error-prone violations should be simple to fix, and shouldn't require long suppression lists.

### ADDING INITIAL SUPPRESSIONS

Error-prone supports suppressing violations in many ways. We would be mainly using **`@SuppressWarnings`** for suppressing violations. Occasionally we would also use command-line flags for suppressions.

### RESOLVING ERRORS AND SUPPRESSIONS

Most of the errors will be fixed on the spot. False positives will be annotated with **`@SuppressWarnings`**. Error prone also offers patching to make the developer's work easier. It suggests the possible solution for that error, while we can manually fix that error, error-prone can also be used to modify the source code with the suggested replacements.

## ARCHUNIT USE CASES

The typical use cases ArchUnit mentions are-

- Package Dependency Checks
- Class Dependency Checks
- Class and Package Containment Checks
- Inheritance Checks
- Annotation Checks
- Layer Checks
- Cycle Checks

For Checkstyle some of the uses cases could be:

- Example 1-

  A class should reside in the proper package, classes name ending with Check should reside the checks package or in any of its sub-packages.

```
@ArchTest
public static final ArchRule archRule = classes()
        .that()
        .haveSimpleNameEndingWith("Check")
        .should()
        .resideInAPackage("com.puppycrawl.tools.checkstyle.checks.*")
        .orShould()
        .resideInAPackage("com.puppycrawl.tools.checkstyle.checks");
```

  A similar rule can also be formed for tests.

- Example 2-

  #4845 mentions test inputs being completely standalone, though this can be achieved by IllegalImport Check, it might still use classes within the same packages as an import statement is not required for them. ArchUnit detects those use cases also as it works on java bytecode.

```
@ArchTest
public static final ArchRule archRule = noClasses()
        .that()
        .haveSimpleNameStartingWith("Input")
        .should()
        .dependOnClassesThat()
        .resideInAnyPackage("..com.puppycrawl.tools.checkstyle..");
```

But as ArchUnit works on bytecode and non-compilable resources are not compiled, it won't be able to check those classes.

One ArchUnit rule is already present in the codebase.

## SUPPRESSING VIOLATIONS

Out of the box ArchUnit supports suppressions by ignoring complete files, files that should be ignored are added in ***archunit_ignore_patterns.txt,*** an example of it would be-

```
archunit_ignore_patterns.txt
```

```
.*some\.pkg\.LegacyService.*
```

But fine control over suppressions is also possible. There are multiple ways of doing it, all of these modify the test itself, I am mentioning a few examples below-

### Example 1- Filtering the locations to be imported.

```
class ExcludeCertainLocations implements ImportOption {

   @Override
   public boolean includes(Location location) {
      return !(location.contains("/classes/"));
   }
}
```

And then using this import option as follows-

```
@AnalyzeClasses(packages = "com.puppycrawl.tools.checkstyle",
       importOptions = ExcludeCertainLocations.class)
```

Example 2- Using a custom location provider-

```
class MyLocationProvider implements LocationProvider {
   @Override
   public Set<Location> get(Class<?> testClass) {
      return Locations
         .ofPackage("com.puppycrawl.tools.checkstyle")
         .stream()
         .filter(location -> !location.contains("/classes/com/puppycrawl/tools/checkstyle/"))
         .collect(Collectors.toSet());
   }
}
```

This can also be used in the AnalyzeClasses annotation as-

```
@AnalyzeClasses(locations = MyLocationProvider.class)
```

Example 3 - Modifying `that(..)` clause to only include certain locations.

```
@ArchTest
public static final ArchRule archRule = noClasses()
   .that(new DescribedPredicate<>("SomeCustomRule") {
       @Override
       public boolean apply(JavaClass input) {
          return "SomePackage.SomeClass".equals(input.getFullName());
       }
   })
   .should()
   .be(yourCustomLogic);
```

## SETTING UP CHECKER FRAMEWORK

The Checker Framework is a pluggable type checker for java that enhances the already present type checking system in java. Checker Framework includes compiler plug-ins ("checkers") that find bugs or verify their absence.

It is a verification tool that **guarantees** that certain types of runtime exceptions won't be caused. It can issue false positives which can be suppressed or the code design can be changed. The good part is that it doesn't miss any case, i.e., no false negatives.

The Checker framework depends upon annotations to verify whether the behavior of code is as expected or not. Our codebase at present is not annotated which is a requirement for using the Checker framework effectively.

Below is a week-wise division of work that will be done to introduce the Checker framework.

- Week 1: Set up Checker framework using Maven. Identify important and relevant checkers as there are checkers for SQL injection which aren't useful to us. Explore documentation of checkers and go through the documentation of various type inference tools the Checker framework suggests. Add exclusions initially to suppress all the violations. Checkers will be added step by step to keep the work focused.

- Week 2 onwards: After successfully setting up the Checker framework, annotate the Checkstyle codebase on a module-by-module basis and resolve all the suppressions reported by the Checker framework. A top to bottom approach will be followed to annotate the classes, classes up in the hierarchy will be annotated first.

Many challenges will possibly occur while introducing the Checker framework:

1. Annotating libraries - When code uses a library that is not currently being compiled, the Checker Framework looks up the library's annotations in its class files. Even if the external library is currently being compiled and it doesn't have any annotations then also the checker can issue false positives.

   From the Checker framework-

   "*If code uses a library that does not contain type annotations, then the type-checker has no way to know the library's behavior. The type-checker makes conservative assumptions about unannotated bytecode. These conservative library annotations invariably lead to checker warnings.*"

   The only solution to this is to either annotate the library being used or just suppress those violations or skip a particular file. Checker framework has already annotated some libraries which are mentioned here.

2. Annotating our codebase – This is a big task that will be split into smaller sub-tasks. The Checker Framework is nice enough to supply some libraries (in-built and external) to automate type inferencing.

   Though some libraries guarantee that all inserted annotations will be correct, still manual verification needs to be done as our codebase might contain some bugs and the annotations might be correct for the codebase. Also, more annotations might be required to make the checker not give a false positive.

From the AnnotateNullable tool of the Daikon invariant detector-

"*The AnnotateNullable tool automatically and soundly determines a subset of the proper @Nullable annotations, reducing the programmer's burden. Each annotation that AnnotateNullable infers is **correct**. The programmer may need to **write some additional** @Nullable annotations, but that is much easier than writing them all*."

3. Checker Framework is slow – Being a compiler plugin, it has to do all the same work as the compiler does, such as resolving overloading and overriding, inferring generics, and type-checking.

   From Checker Framework-

   "*Using a pluggable type-checker increases compile times by a factor of **2–10**. Slow compilation speed is probably the worst thing about the Checker Framework*."

   The Checker Framework mentions some workarounds to improve the performance in the performance improvement section.

## INITIAL RESULTS AFTER ACTIVATING CHECKER FRAMEWORK

Checker Framework was activated to see how much work actually has to be done and how will it be broken into subparts.

It was only enabled with just the nullness checker and it reported **621** errors. The full log can be found here.

The initial plan would be to just introduce one checker, i.e. nullness checker, and fix violations from it, other checkers can also be introduced after successfully introducing the nullness checker to the project.

There is a lot of work to be done, but this is how it will be broken into subparts-

### ADDING INITIAL SUPPRESSIONS

Checker framework has many ways to suppress warnings, all of them can be found at suppressing warnings.

The standard approach followed in our project would be to modify the pom.xml and add a command-line argument to suppress warnings from a particular class or package. This behavior can also be extracted to a shell script (just like we have for pitest).

```
<compilerArgs>
   <arg>-AskipDefs=^com\.puppycrawl\.tools\.checkstyle\.checks\.design\.FinalClassCheck</arg>
   <arg>-AskipUses=^com\.puppycrawl\.tools\.checkstyle\.checks\.design\.FinalClassCheck</arg>
</compilerArgs>
```

This would be present in the profile of Checker Framework. The actual changes in pom.xml can be found here.

Apart from doing bunch suppressions, we can suppress warnings individually using the `@SuppressWarnings` and `@AssumeAssertion`.

Suppressions will be resolved one by one. PR for all the modules issuing warnings will be made.

The code will be annotated (for nullness checker) using AnnotateNullable. I tried to use the tool to automatically add annotations but am unsuccessful till now, the documentation is quite old and many resources are not available on how to use it, I have raised daikon/381, and hopefully, someone will help us out, if not, I will mail the maintainers personally, requesting for some help. If nothing works out then we can proceed toward the manual annotation of the code or use IntellijIdea for adding annotations.

We would have to change our usual way of coding. We would have to insert annotations like `@Nullable` and `@NonNull` while writing any piece of code. This practice would have to be followed by every programmer. It might sound tough to annotate everything but Checker Framework has built-in automatic type inference so annotating every variable or field won't be required, Checker framework will issue warnings and we can annotate our code to get rid of those warnings or permanently suppress those warnings.

## WHY ME?

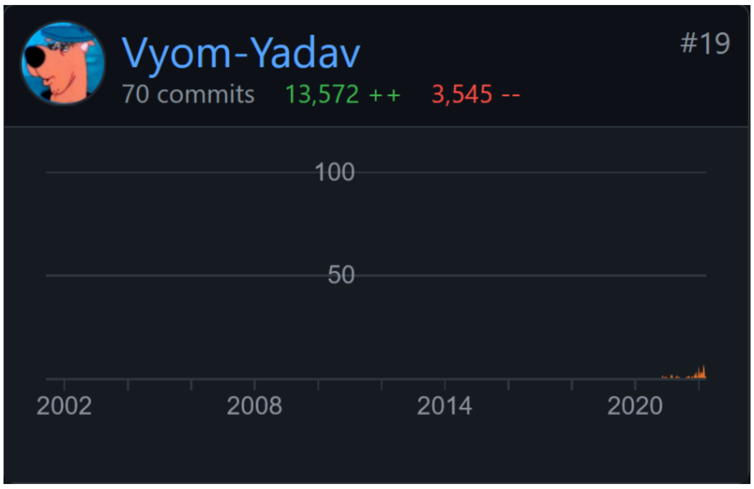
Good question, I have been very active in the community. I started contributing to the organization in November 2020. My first commit was on 27-Nov-2020. Since then, I have been around the community, fixing bugs, improving documentation, adding new features, and a lot more things.

During GSoC, I will be able to dedicate ~30 hours a week to the project.

I am having my end-semester exams in May, but they will most probably finish within the first or second week of the community bonding period.

Since this is a large project, with an extended coding period, I will be having my college examinations and practicals in between, somewhere in September (dates not announced yet). These things won't hamper the project, many leaves won't be taken.

### COMMITS MADE TO THE PROJECT



[All the commits made to the Checkstyle project]()

Here is a list of PRs that I made to the project –

## DOCUMENTATION CHANGES

- Issue #7564: Added example for default config
- Issue #7567: Adding code examples for default configuration of MethodNameCheck
- Issue #7652: Updating doc for NestedIfDepth
- Issue #9131: Adding example for AvoidEscapedUnicodeCharacters
- Issue #7583: Added Code example for default config of NeedBraces and …
- Issue #9581: updated example of AST for TokenTypes.TEXT_BLOCK_CONTENT

Documentation is also updated when any new feature is introduced or when a crucial bug is fixed, etc.

## BUG FIXES

- Issue #5779: fixing no violation on parenthesis in if statement betwe…
- Issue #8237: fixing no whitespace violation in empty constructors
- Issue #9357: FinalClass now exempts private-only constructor classes that also have nested subclasses
- Issue #9941: Fixing AtclauseOrder falsely ignoring method with annotation
- Issue #9957: Fixing Unnecessary Parentheses false positives
- Issue #10756: JavadocMetadataScraper now correctly identifies check name
- Issue #10810: Fixing StackOverflow error in AbstractExpressionHandler
- Issue #10835: Fixing default access modifier for CheckUtil

- Issue #10901: Updated org.reflections:reflections dependency to new version 0.10.1 and fixed NPE's
- Issue #11015: Fixed false negative about ternary operator in SimplifyBooleanExpressionCheck
- Issue #11038: ParameterAssignment now detects problems for Lambda parameters
- Issue #11030: Enhanced logging approach of SummaryJavadocCheck and used AST-based approach to get content of inline tag.
- Issue #11031: Fixed false negative about concatenated strings in StringLiteralEqualityCheck
- Issue #11061: False negative around TokenTypes.LOR in UnnecessaryParenthesesCheck
- Issue #6320: Kill surviving mutations for REMOVE_CONDITIONALS in UnusedLocalVariableCheck
- Issue #11268: Fix false negative in RedundantModifierCheck
- Issue #11213: Fixed false positive in SummaryJavadocCheck

## NEW FEATURES

- Issue #7504: New Check UnusedLocalVariable
- Issue #11116: Added ELLIPSIS support to WhitespaceAfterCheck
- Issue #11259: Add record support to RedundantModifier
- Issue #11298: Added switch statements and lambda support to WhitespaceAfter
- Issue #10907: Modify LineLengthCheck to support validating import and package statements

## INFRA ISSUES

- Issue #10943: Message validation in our Inputs should not have trailing and leading '.*'

- Issue #10752: Added support for violation N lines above/below in Inline Config Parser
- Issue #11091: Fail test execution when input file hasn't specified violation messages
- Issue #11163: Enforced file size on inputs
- Issue #10737: Migrate to inline config parser for SuppressWithNearbyCommentFilter
- Issue #10737: Migrate to inline config parser for SuppressWarningsFilterTest
- Issue #10737: Migrate to inline config parser in SuppressWithPlainTextCommentFilterTest
- Issue #10737: Migrate to inline config parser for SuppressionSingleFilterTest
- Issue #11438: Converted test for SuppressionXpathFilterTest to use inlined config in Input files
- Issue #11439: Converted test for SuppressionXpathSingleFilterTest to use inlined config in Input files
- minor: Migrate missed out test to InlineConfigParser in SuppressWithNearbyCommentFilterTest

## MISCELLANEOUS

- Issue #10064: Refactored XDocsPagesTest and AllChecksTest to reduce CognitiveComplexity
- Issue #10747: fixing JavadocMetadataScraper to not fail silently if it can't write the XML files
- Issue #10747: MetadataGeneratorUtil now reports errors back to the users
- Issue #10910: Removed curly braces usage for javadoc tags in translations
- Issue #11194: Bump year to 2022
- Issue #11215: add file filters for non-compilable files in openjdk16

- Issue #11281: Removed extra checkstyle build in no-exception-test.sh
- Issue #11201: Moved methods common in UnusedLocalVariableCheck and FinalClassCheck to util
- checkstyle/contribution minor: add new check UnusedLocalVariable
- minor: Remove outdated comment from UnusedLocalVariable
- minor: Wrapped chained method calls
- minor: Remove closed issues that are still referenced in files
- minor: Remove closed issues that are still referenced in files
- minor: remove duplicate exclusion in openjdk17-excluded.files
- checkstyle/contribution minor: exclude non-compilable files in openjdk17

## AUTOMATION AND CI

- Issue #11194: Automate bumping license year
- Issue #11442: Check no closed issue references workflow is now triggered on pull requests
- minor: Add required permissions to bump_license_year.yml

## OPEN PR'S

- Issue #11087: New check ChainedMethodCallWrap
- Issue #11365: Fixed false positives related to anonymous inner classes in FinalClassCheck

## AFTER GSOC

Working on the Checkstyle project taught me a lot of things, I will be forever grateful to everyone who helped me in my journey so far.

I will certainly not disappear after GSoC, I wish to be a part of the maintenance team of Checkstyle in the future.

The pace of contributions may fluctuate but I will be there to fix issues, assist newcomers and help the project grow.

I have attached my resume alongside the proposal.

Looking forward to GSoC 2022!