Elliott de Bruin
Ofek Inbar
Daniel Wang
Jessica Chen
CSE 403

Flint

A Programmable Style and Documentation Linter for Java

When developing and maintaining a clean and readable codebase, having style guides is crucial. Style guides lay out a set of rules for how contributors should format and document their code. In most cases the set of style rules for a codebase is too large for a programmer to have memorized, so static analysis tools, also known as linters, are used to review code for violations of style. However, many of these tools are very rigid in their configuration of how and when to check for which style rules, and there is much room for improvement in programmability. Our tool, Flint, is designed to address these inadequacies. Flint is a programmable style linter that can be used as either an IDE plugin or command line tool for Java code.

Currently one of the most used static analysis tools for Java code style is Checkstyle. Checkstyle allows the user to define a set of style guidelines from premade rules as well as custom rules for their code. Although Checkstyle has features similar to Flint, it has limitations that make it not the most ideal tool to use. One limitation is that Checkstyle is offered natively as a Command Line Tool. While this may make it easy to use for people who do not program in IDE's it requires the developer to finish writing their code before they run the linter. This can lead to a loss in productivity where the user can be caught in a loop of writing code, running the tool, going back and fixing the code, running the tool again, and so on. A solution to this is to offer the linter as plugin or extension to a popular IDE, so that it can be run on a file whenever it is saved (thus saving the developer a context switch). However, it should be noted that this issue alone is not overly deterring, as there are numerous 3rd-party client applications for Checkstyle that are offered as IDE extensions. While it would be preferable for this functionality to be offered natively, the issue is mitigated by the existence of these client extensions.

The larger, more serious issue is that the way in which Checkstyle is configured is not customizable enough. While Checkstyle's XML configuration allows you to define which checks to run for a specific project, and even allows for parameters to be passed to them, it simply cannot match the amount of control that a programmable configuration could offer. If the configuration was instead written in Java, then users would have the option of running checks conditioned on certain situations or outcomes, and in general they would have much more control over the flow of logic through the checks that are run. If, for example, a user wished to perform a time-consuming check only for files that are under a certain line limit, Checkstyle would require them to factor this logic into their check. It is clearer and more intuitive if the configuration simply offered some way to run checks only under certain conditions. Flint is designed in such a way that the configuration file (and indeed every other part of the tool as well) is written in Java, so the user does not have to context-switch between different languages and has more fine-grain control over the tool.

Another commonly used style checking tool is Google's java style formatter, google-java-format. This tool can be run from the command line or used as an IDE plugin. Google-java-format checks Java code for style violations and automatically reformats the code as errors are found. This is very helpful for Java programmers that follow Google's style guidelines, but the downside of google-java-format is that it only enforces Google's style guide and is not easily customizable. For Java programmers that follow different style guides, Flint will be the more useful tool as it will allow users to customize rules to closely fit their needs, whether it be Google's style guidelines or a project specific one.

The key part of creating Flint will be creating the rules that analyze given Java code. Our approach to designing this software is to provide a framework that can run in response to a file-save and handles any issues that can come of reading and parsing a Java file, as well as provide an explicitly-defined method to create custom rules to check files against as well as a way to configure how to run them. We will also use this approach ourselves to develop a set of rules that we can ship with the application itself, to both demonstrate examples to the users and also to save them the time and effort of implementing some of the more common rules. Users will be able to use these rules in their configuration or they can extend from our abstract rule class and create their own rules that cater to their specific needs, all within Java. The benefits of this approach is that it allows users the maximum control in both defining their own rules and also configuring how to run them. Resultantly, we hope to enhance the developer experience by allowing them to configure the tool to function in exactly the way that will boost their productivity the most.

Increasing developers' productivity is one of our main motivations for creating Flint. For that goal to be realized we need to address challenges and pain points that hinder a tool's ability to help developers and push them away from the tool. In a study done by Microsoft (Christakis) on what developers want and need from program analysis, they found some of the things push developers away from a program analyzer the most are having wrong checks are on by default, bad warning messages, too many false positives, difficult to fit into workflow, and bad visualization of warnings. While developing Flint we will be focusing on ensuring that our users do not face these difficulties, as well as many others, with our product so we can make their programming experience as efficient and smooth as possible.

The limitations of our approach may be that we will provide less pre-built rules and it may be more difficult for some users to learn how to create their own custom rules as opposed to existing solutions such as Checkstyle. There are also limitations on the kinds of violations a rule can check for. For example, one of Google's style guide rules defines a requirement involving static imports of specific kinds of classes. This requires referencing multiple files in one rule, which will not be possible given our current approach (this functionality is not supported in Checkstyle either).

A sample configuration for a project is provided below. This configuration counts the lines in a file, runs a rule, and the runs a resource expensive rule conditioned on the line count. This would demonstrate the flexibility of using java to define lint rule, namely only run resource intensive checks

(like, say, identifying sections of repeated code of any length) under certain conditions. This sort of flexibility is either impossible or exceedingly difficult/unintuitive to provide in XML.

```
public class FlintConfig18 extends FlintConfiguration {
  @Override
  public Collection<LintFailure> runChecks(RandomAccessFile inputFile, CompilationUnit astRoot) throws IOException {
    int lineCount = 0;
    while (inputFile.readLine() != null) {
      lineCount++;
    }
    inputFile.seek(0);

    Collection<LintFailure> result = new HashSet<>();
    result.addAll(TabsNotSpacesRule.run(inputFile, astRoot));
    if (lineCount < 500) {
      inputFile.seek(0);
      result.addAll(ResourceExpensiveRule.run(inputFile, astRoot));
    }
    return result;
  }
}
```

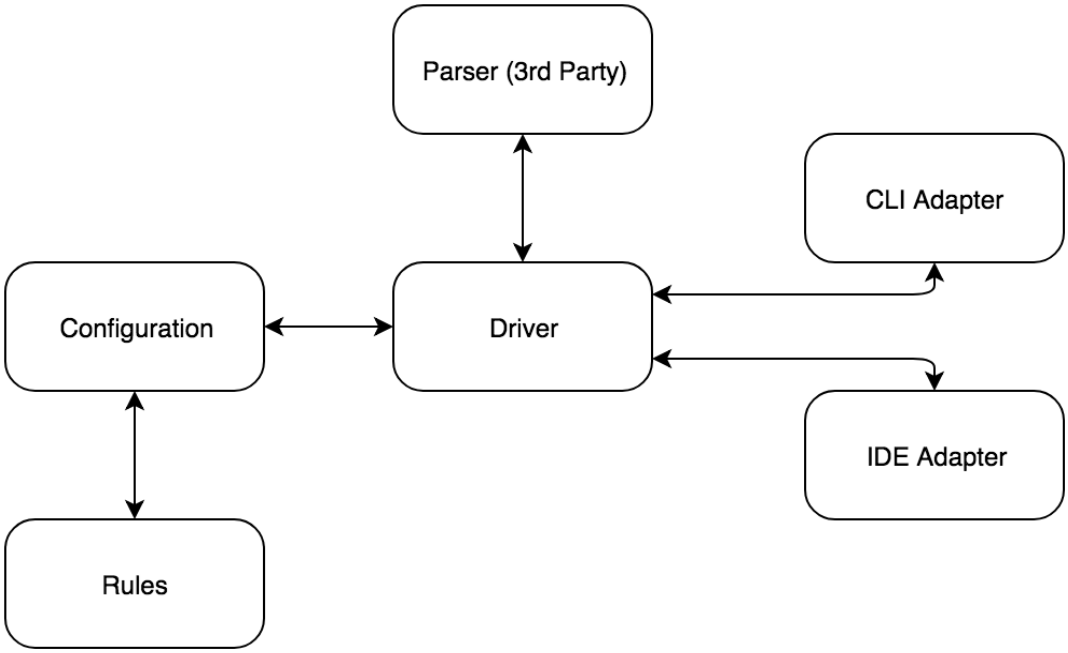
Challenges and Risks

The single largest challenge we see with developing Flint on schedule is implementing the customization feature such that it is easy to understand and use from the first time and it is able to be used to create any rules that a client requires for their project. In order to properly develop Flint on time, we will start by creating the general rule class and then we will build each built-in rule by extending the general class. By building each premade rule the same way a client will make a custom rule we will be able to learn the best process, what is needed in a rule class, and what is not necessary. This way by the time we have finished the premade rules we will have refined the process and be able to make it user-friendly and effective for all new rules.

User Testing

One crucial aspect of creating Flint will be conducting and reviewing useful and informative experiments to help develop and refine the tool. We will conduct two distinct rounds of User Testing, once after the key features are implemented (running on file-save, custom rule creation) and once after the product is basically complete. This will help give us guidance on how best to continue developing the tool and whether, in the end, we have built a useful product. User Testing will consist of asking Java programmers to use Flint on one of their projects and to create new style rules for their projects, using only the user manual and documentation that will be available to all new users. Ideally, we want users to be able to use Flint with ease from the first use, so seeing how new users interact with the software will allow us to refine the features of the tool as well as the user manual. We will measure success on two fronts: 1) User feedback on how easy it was to figure out how to use the main features of the tool, and 2) User feedback on how helpful/useful the tool was, and how likely they would be to use it in the future. If these experiments succeed, we will have received useful input from real users on which to base our final product.

Architecture Design



CLI/IDE Adapter:

- Responsible for triggering Driver, and provides the driver with a filepath of the file to lint and an instantiated FlintConfiguration object (written by the user)
- Receives collection of Failure objects (which each define a startLine, endLine, startCol, endCol, and message) from Driver and outputs it (either to the IDE GUI or to the console)

Driver:

- Receives “filepath” and FlintConfiguration object from Adapter
- Responsible for checking that the file at “filepath” is a Java file and that it compiles
- Responsible for parsing the file at “filepath” and passing the resulting Abstract Syntax Tree, along with a Scanner to the file at “filepath” to a “run” method in the FlintConfiguration object
- Receives a collection of Failure objects from FlintConfiguration and passes it on to the Adapter

Configuration:

- Receives a Scanner to the file to lint and an AST (Abstract Syntax Tree) from Driver
- Defines how to run whatever rules it chooses to run (in what order, under what conditions, etc), and passes the “filepath” and AST to each rule
- Receives a collection of Failures from each check, compiles them into 1 collection and passes it onto the Driver

Rules:

- Receives a Scanner to the file to lint and an AST from Configuration
- Analyzes the file however it chooses and returns a collection of Failures (empty if there are none) to Configuration

Implementation Plan

The first stage of our implementation plan is to develop a CLI version of Flint with the most important features, which are triggering the driver, passing the file path to the driver, parsing the file data, and creating an instantiated configuration object. For the CLI the user will interact with Flint by running the CLI, and passing in the file path to the file they want analyzed and their configuration file. An example command could be:

```
flint -file-path src/main/java/star/Star.java -config-path src/main/java/star/
-config-class FlintConfig403
```

For each error found by the Flint CLI, it will output the line numbers that the error begins and ends on, the column numbers where the error begins and ends, and the message for what error was found. The second stage of our implementation plan is to use the base of our CLI tool to build a Flint plugin for Intellij Idea. For the plugin the user will create a configuration file named FlintConfig18.java that will stay in the root of the package for the project the user is working on. The user will only be able to interact with Flint by editing their configuration file. Then on each file save Flint will first check if the file compiles and if it does it will run checks for each rule in the configuration file. For each error found in the given file Flint will underline the error in yellow and they will be able to hover over to underline to display the error message.

We foresee using several tools and technologies while completing this project. We will use **Java** to write the tool and **Gradle** to build the project, as it is the build system we are most familiar with. We will also be using **JUnit** to run our automated test suite. We will use **JavaParser** to parse Java files. Later in the quarter, we will use the **IntelliJ Idea** plugin API to build our IDE Adapter, and we will use the IDE to test our plugin.

- Ofek - Build parsing functionality of driver, test adapters
- Daniel - Build CLI/IDE adapters, test compile-check functionality of driver
- Elliott - Build compile-check functionality of driver, test default rules
- Jessica - Build abstract classes for Configuration and Rules as well as some default rules, test parsing functionality of driver

Bibliography

"Checkstyle – Checkstyle 8.17." *Checkstyle*, 27 Jan. 2019, checkstyle.sourceforge.net/.

"google/google-java-format: Reformats Java source code to comply with Google Java style." *Google*, 10 Jan. 2019, github.com/google/google-java-format/.

Christakis, Maria, and Christian Bird. "What Developers Want and Need from Program Analysis: An Empirical Study." *Microsoft.com*, Microsoft Research, www.microsoft.com/en-us/research/uploads/prod/2016/07/What-Developers-Want-and-Need-from-Program-Analysis-An-Empirical-Study.pdf.

Week-by-Week

Jan 28 - Feb 1	~~● Research IDE plugin/extension development (decide on an IDE to develop for)~~ ~~● Design architecture & implementation plan (produce deliverables)~~ ~~● Write up specifications & user manual~~
Feb 4 - Feb 8	~~● Write tests for Syntax Parser~~ ~~● Integrate 3rd Party Syntax Parser package~~ ~~● Test Driver~~ ~~● Driver Fully Built~~ ~~● Test CLI Adapter~~ ~~● CLI Adapter Fully Built~~
Feb 11 - Feb 15	~~● Test customization support~~ ~~● Build customization support~~ ● User testing
Feb 18 - Feb 22	● Test IDE Adapter ● IDE Adapter Fully Built ● Test file-save response ● Build file-save response
Feb 25 - Mar 1	● Test default rules ● Develop default rules ● Test UI feedback (red squiggly lines) ● Build UI feedback
Mar 4 - Mar 8	● Final touch-ups
Mar 11 - Mar 15	● User testing
Mar 18 - Mar 21	● Prepare presentation

20 hour spent on this proposal.