Elliott de Bruin
Ofek Inbar
Daniel Wang
Jessica Chen
CSE 403

Flint

A Programmable Style and Documentation Linter for Java

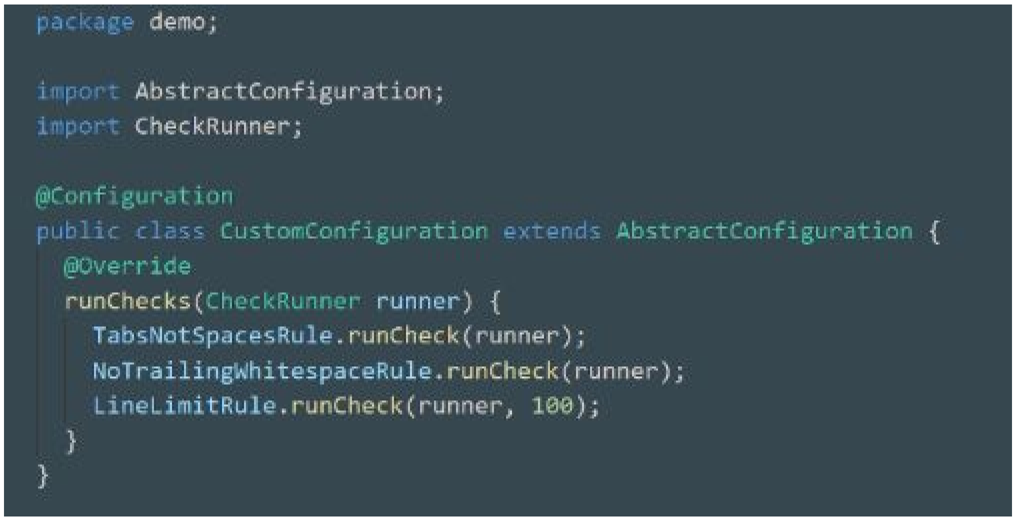
When a team with many contributors is developing and maintaining a clean and readable codebase, having style guides is crucial. Style guides lay out a set of rules for how contributors should format and document their code in order to keep the code base uniform and readable to everyone. In most cases 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 and coding rules. Most linters however need to be run as a Command Line Tool, which results in programmers finding many bugs or warnings after they have finished writing their code. Our tool, Flint, can be used to solve these problems and aid developers. Flint is a programmable style linter IDE plugin for Java code.

While many style linters are not run as you are writing your code, Flint runs on every file-save, just like the compiler, and will automatically point out issues in code as it is being written. Flint will come with a built-in set of common and uncommon style rules that are useful for many programming projects. While these built-in rules will cover many style rules, we cannot account for project specific rules so Flint allows you to implement your own rules that perfectly fit your needs. Rule configuration and custom rule creation will be defined completely in Java for ease of use for Java developers. Another feature Flint will have is the ability to mark temporary comments. When writing code many software developers will leave notes to themselves as inline comments, but these comments may not always fit in with the style guidelines and will need to removed once the code is completed. To help developers with this problem, when using Flint the user will be able to mark comment notes as temporary. Comments marked as temporary will be highlighted for ease to find and remove at a later point and will not be analyzed during static analysis.

Currently one of the most used static analysis tools for Java code style is Checkstyle. Checkstyle is a static code analysis tool that is used to automate the code review process. 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. To configure which rules to use when linting a particular project, you must write a configuration file in XML format. When working on a project that is in Java a developer may not be familiar with writing XML and will have to learn how to do so before writing any configuration for Checkstyle. Flint on the other hand will allow the user to extend a generic Java configuration class and implement define their project's custom configuration in Java, the language they are already working in. Implementing the customization feature like this will allow developers to easily move between writing code for their project, and writing new rules. Checkstyle is offered as a Command Line Tool as well as an IDE extension, which allows the checker to run even the user is still working. Flint will offer this same usability because having Flint check for and mark style violations as the user writes code will greatly increase productivity. Checkstyle also has a limitation that any style rule applied can only be applied to a single file at a time, so a single rule may not reference two separate Java files. Our goal is to allow Flint to support multi-file rules. Another major con of using Checkstyle is that it only checks for style violations but does not automatically reformat the code to fix them. We plan on developing Flint to check for style violations as well as reformat the code to fix the violation. Having this feature will save the user much more time than Checkstyle does, allowing them to be more productive.

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, specific to their projects, Flint will be the most useful tool for them. Flint will allow users to customize rules to closely fit their needs, whether it be Google's style guidelines or a project specific one, and will automatically fix style violations.

The key part of creating Flint will be creating the rules that analyze given Java code. Our approach to designing this software is to start by creating a generic rule class that will examine each line of a given Java file and perform generic checks to make sure that the file is a valid input. Then for each specific style rule that is created for Flint, we will be able to extend the generic rule class and add methods to check for the aspects of the rule we want, such as checking for no trailing whitespace on lines. We will use this approach both to implement the more common rules that will be built into Flint as well as the custom rule feature. Users will be able to extend from the generic rule class and create their own rule classes that cater to their specific needs, all in Java. The benefits of this approach is that all rules will be based off of the generic rule class so the code is being reused for each new rule and if how all the rules analyze or reads a file needs to be changed, it can be done in one place. With this new kind of approach, we may boost the work efficiency and further enhance productivity for future java based development. The limitations of this approach may be that what the client is able to check for depends on how we implement Flint to read files. For example, if Flint were to read only one file at a time, a client would not be able to make a rule for something that would need to consider multiple files.



```
package demo;

import AbstractConfiguration;
import CheckRunner;

@Configuration
public class CustomConfiguration extends AbstractConfiguration {
  @Override
  runChecks(CheckRunner runner) {
    TabsNotSpacesRule.runCheck(runner);
    NoTrailingWhitespaceRule.runCheck(runner);
    LineLimitRule.runCheck(runner, 100);
  }
}
```

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.

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.

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/.

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 main method (scan folder for files to parse) ● Build main method ● Test CLI Adapter ● Build CLI Adapter
Feb 11 - Feb 15	● Test customization support ● Build customization support ● User testing
Feb 18 - Feb 22	● Test IDE Adapter ● Build IDE Adapter ● 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.