Документация по программе nerd-linter

Автор: Viancis valeriy

Содержание

1 Введение

У меня не так много места и времени, прежде чем вы пройдете дальше, пропустив этот пункт, где я могу объяснить важные, как по мне, детали. Проект был выполнен в рамках задания по предмету "Языки программирования".

Начнём с самого начала:

Linter — это инструмент, который выступает в роли строгого преподавателя кода, проверяющего его на наличие ошибок, несоответствий стилю и потенциальных проблем. Он анализирует исходный код, выдавая рекомендации и предупреждения, чтобы разработчики могли создавать более качественные и поддерживаемые программы.

AST (Abstract Syntax Tree) — это структура данных, представляющая синтаксическую структуру программы в виде дерева. Каждый узел дерева соответствует конструкции языка программирования, что позволяет анализировать и манипулировать кодом на высоком уровне, не углубляясь в детали синтаксиса.

Как возникла идея создать именно такой линтер?

У каждого из нас есть такой друг, который может доказывать, что ваш код плохой, потому что он не соответствует каким-то правилам оформления. Так вот, у меня такого друга нет, а надо бы — с моим миллионом ошибок в кодстайле каждого проекта.

Чем отличается этот линтер от своих конкурентов?

Если не вдаваться в детали, то есть два ключевых аспекта. Первое — это расширяемость, второе — парсинг дерева кода до начала проверок. Расширяемость, думаю, понятна: это возможность реализовать интерфейс и в будущем легко добавлять проверки для других языков. Да, абстрактное дерево кода отличается от языка к языку, поэтому написать один единый инструмент крайне невозможно, кто бы мог подумать! Парсинг до проверок предоставляет возможность писать эти проверки, не особо понимая, как работает **AST**. В этой документации я также затрону эту тему.

Будет ли реализация локализации на другие языки?

Да, будет.

Сколько времени ушло на разработку?

Не учитывая время, когда я читал про **AST** для Java, — около двух-трех недель. Может показаться сравнительно недолго, но эти недели я жил проектом и много свободного времени уделял именно ему. Проверок я реализовал немного; на самом деле, мне не так сильно интересно копаться в правильности кодстайла Java, поэтому я реализовал пару проверок на каждую сущность, чтобы показать, что мой парсер кода по **AST** позволяет писать проверки очень легко.

Какие выводы я сделал в ходе разработки программы?

Абстракция — это зло. Рефлексией в Java и Kotlin лучше не злоупотреблять там, где это не нужно, иначе время выполнения будет только увеличиваться, а отлавливать ошибки станет сложнее. **Kotlin** — потрясающий язык программирования, который решает множество проблем, существующих в **Java**.

P.s Код содержит не малое количество несоответствий кодстайлу, много структурных и идейных просчетов. Я лишь хочу сказать, что автор не несет ответственность за соблюдение всех правил.

2 Установка

Следуйте приведённым ниже шагам для установки **nerd-linter**:

1. Клонируйте репозиторий:

   ```
   git clone git@github.com:Vediusse/nerd-linter.git
   ```

2. Перейдите в директорию проекта:

   ```
   cd nerd-linter
   ```

3. Соберите проект с помощью Maven:

   ```
   mvn clean install
   ```

4. Вернитесь в домашнюю директорию:

   ```
   cd ~
   ```

5. Создайте директорию `bin`:

   ```
   mkdir bin
   ```

6. Откройте файл `nerd-linter` в текстовом редакторе:

   ```
   nano ~/bin/nerd-linter
   ```

7. Вставьте следующий код в открывшийся файл:

   ```
   #!/bin/bash
   java -jar путь/nerd-linter/target/nerd-linter-1.0-SNAPSHOT-jar-with-dependencies.jar "$@"
   ```

8. Сделайте файл исполняемым:

   ```
   chmod +x ~/bin/nerd-linter
   ```

9. Откройте файл `.zshrc` для редактирования:

   ```
   nano ~/.zshrc
   ```

10. Вставьте следующую строку в конец файла:

    ```
    export PATH="$HOME/bin:$PATH"
    ```

11. Примените изменения:

    ```
    source ~/.zshrc
    ```

12. Запустите `nerd-linter` с конфигурацией и наслаждаемся:

    ```
    nerd-linter --conf=./nerd-conf
    ```

3 Некоторые сведения о структуре

3.1 Проверки

Вместо того чтобы проходить по всему проекту, я лучше укажу ключевые места, которые понадобятся для добавления нового функционала. Для ориентирования в проекте я буду использовать пакеты Java.

Пакет `checkers.language.название.category.категория` содержит наши проверки. Существует несколько важных аспектов, которые следует учитывать при добавлении новой проверки.

Все проверки делятся на три типа:

- **Errors** - это критические ошибки, которые наверняка приведут к проблемам.
- **Warnings** - это не критические ошибки, которые просто лучше поправить.
- **Conventions** - это тоже не критические ошибки, которые просто лучше поправить, но как правило это соглашения, которые я вообще не знаю для кого нужны.

Кроме того, у каждой проверки есть специальная нумерация. Например, `WJ03`, где первая буква указывает на категорию проверки, вторая — на язык, а затем следует порядковый номер. Соблюдение этого формата не обязательно, но будет весьма полезно.

Чтобы зарегистрировать новую проверку, необходимо добавить аннотацию к классу:

```
@CodeCheckInfo(code = "WJ03", level = Category.WARNING)
```

Затем следует унаследовать класс от `BaseCodeCheck()` и реализовать метод `check`, который вернет нам объект `Response`. Рекомендуется использовать конструкцию `return createCheckResponse(errors)` для возврата объекта `Response`.

Стандартные проверки для Java:

- **WJ01** Проверка на наличие аннотации `@Override` у переопределяемых методов. Выдаётся предупреждение, если метод, переопределяющий родительский, не помечен аннотацией `@Override`.
- **WJ02** Проверка порядка полей в классе. Выдаётся предупреждение, если поля класса не упорядочены по заданному критерию (например, статические поля, затем финальные и инстанс-поля).
- **WJ03** Проверка модификаторов видимости. Выдаётся предупреждение, если у полей или методов класса не указаны модификаторы доступа (или они равны `null`), и используется значение по умолчанию.
- **WJ04** Проверка на наличие геттеров и сеттеров. Выдаётся предупреждение, если публичное поле класса не имеет соответствующего геттера или сеттера.
- **WU01** Проверка на наличие комментариев. Это личная боль и моя личная проверка, мне не нравится впринципе идея inline комменатриев.
- **WU02** Проверка на количество символов, исправлением должен заниматься форматер, но пока что его нет.
- **EJ01** Проверка отсутствия инструкции `return`. Выдаётся предупреждение, если метод с возвращаемым типом не имеет инструкции `return`.
- **CJ01** Проверка порядка импортов. Выдаётся предупреждение, если порядок импортов нарушен (например, импорты из пакетов `java` или `javax` идут после других импортов).
- **CJ02** Проверка соглашений об именовании. Выдаётся предупреждение, если имя класса, поля или метода не соответствует установленным соглашениям об именовании (например, классы должны использовать `CamelCase`, поля и методы — `camelCase`, константы — $\texttt{UPPER}_S NAKE_C ASE$).

3.2 Сущности кода

На уровне пакета `checkers` и `checkers.language.java.entity` находятся сущности кода, которые описывают код на каком-то языке программирования. Каждая сущность представляет собой абстрактный класс, от которого могут наследоваться более конкретные реализации. Ниже приведены основные сущности кода и их назначения:

- **CodeEntity**: Абстрактный класс, представляющий общую сущность кода. Содержит основные атрибуты: имя, тип и узел дерева разбора.

- **VariableEntity**: Описывает переменные в коде. Включает информацию о имени, типе и узле дерева разбора. (локальные переменные и аргументы функций)
- **FunctionEntity**: Представляет функции в коде. Содержит параметры функции и локальные переменные. (функции и методы для удобства назвал именно function)
- **FieldEntity**: Описывает поля в классах или структурах. Хранит информацию о имени, типе, узле дерева разбора и инициализаторе.
- **ConstantEntity**: Представляет константы в коде. Содержит имя и тип, который по умолчанию устанавливается в "constant". (константы)
- **BinaryExpressionEntity**: Описывает бинарные выражения в коде. Хранит операнды и оператор, используемый в выражении. (бинарные операции. Является вспомогательным для FieldEntity)
- **StructureEntity**: Описывает структуры, которые могут содержать поля и функции. Имеет списки для хранения полей, функций и вложенных структур. (Наши классы)
- **FileEntity**: Описывает файлы, содержащие код. Хранит имя файла и узел дерева разбора. Используется для представления файлов в системе, содержащих различные сущности кода. (Общая сущность, которая описывает целый файл)
- **ImportEntity**: Представляет импортируемые файлы или модули в коде. Хранит имя импортируемого файла и узел дерева разбора. Позволяет управлять зависимостями между различными модулями и файлами в проекте. (Это то, что FileEntity импортирует)

Теперь обладая такой абстракцией, можно написать свой CodeEntity класс для каждого языка, который опишет логику языка программирования.

3.3 Анализатор AST

На уровне пакета `checkers.language.язык` уже может существовать анализатор AST, или его необходимо добавить. У него есть методы, которые работают с AST деревом и составляют один единный FileEntity. Служит этот слой, чтобы в проверках максимально уйти от парсинга дерева кода и сосредоточиться на самих проверках. Названия методов интуитивно понятны, поэтому описывать за что ответственен каждый из них не буду.

3.4 Процессинг FileEntity

Анализатор AST работает с каждым файлом отдельно, что ограничивает его возможности в получении информации о содержимом импортов или классах, от которых наследуются текущие классы. На этапе анализа AST используются заглушки, что позволяет продолжить процесс без необходимости получения полной информации о зависимостях.

Во время процессинга мы анализируем результаты, полученные на предыдущем этапе, и извлекаем необходимые сущности, чтобы подтянуть соответствующие зависимости. Это обеспечивает полное представление о структуре кода и его взаимосвязях.

3.5 В общем по структуре

nerd-linter построен так, чтобы человек не знающий AST мог с легкостью писать новые проверки, не особо думая про различие в языке или в AST. Очень хотелось бы иметь проверки в декларативном стиле, но это можно реализовать в отдельном модуле.

4 TODO:

Проект nerd-linter открыт к вашим issue и pull request по добавлению новых проверок или правкам к уже существующим. У меня уже есть некоторые идеи по улучшению проекта:

- Локализация - в папке template разбить всё по папкам и создать сервис Template, который принимает язык, номер ошибки и язык пользователя и отдает соответсвующий текст из jinja2.
- Анализаторы AST для других языков - тут важно, что treeSitter, который я использую, для постройки дерева кода, строит такие узлы, что сделать какой то универсальные парсер не получится даже в теории. Поэтому на каждый язык по одному анализатору.
- Форматтер для общих правил - убрать все комментарии, сократить длину строки – лучше не ругать пользователя за это, а самому убирать такие душные моменты.
- Декларативный стиль проверок - возможность уйти от логики работы даже с entity. Позволяет динамично патчить приложение.