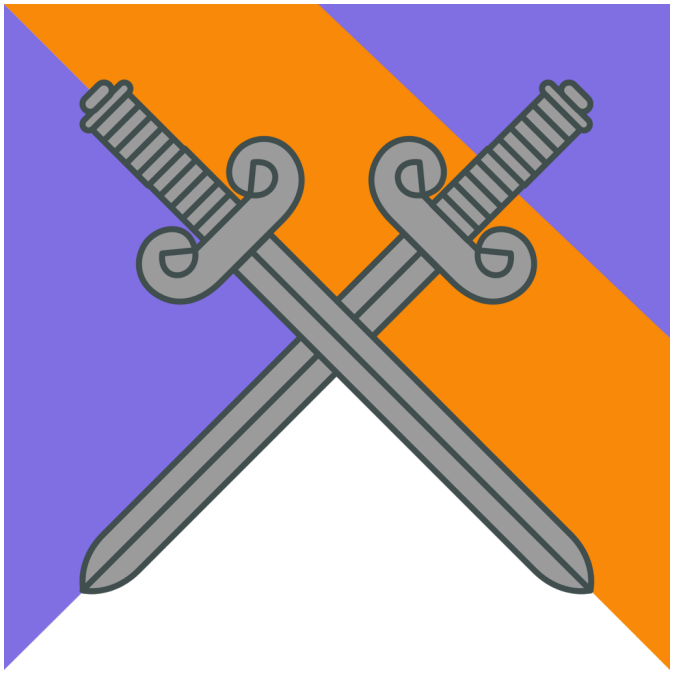


November 9, 2020

Contents

1 Introduction

It is necessary to conform to a specific style of code during software development, otherwise it will reduce the ability to better understand programmers' intent and find more functional defects. Analyzers, in turn, have methods for finding and correcting style errors.

Static code analysis is useful not only for optimization and increasing effectiveness but also for automatic error detection.

There are many methods and techniques used by existing analyzers to find bugs(path-sensitive data flow analysis [**?**], alias analysis (Wu et al., 2013), type analysis (Wand, 1987), symbolic execution (Slaby, 2013), value flow analysis (Sui and Xue, 2016), abstract interpretation (Slaby, 2013)).

Static analysis can be thought of as an automated code review process. Of the tasks solved by static code analysis programs, two main ones can be distinguished: identifying errors in programs and recommending code formatting. That is, the analyzer allows you to check whether the source code complies with the accepted coding standard. Also, a static analyzer can be used to determine the maintainability of a code, which is how easy it is to analyze, modify and adapt a given software. Static analysis tools allow you to identify a large number of errors in the design phase, which significantly reduces the development cost of the entire project. Static analysis covers the entire code - it checks even those code fragments that are difficult to test. It does not depend on the compiler used and the environment in which the compiled program will be executed.

2 Definition

Before continue, it is necessary to define some terms so that the reader correctly understands the context of what was written. The first and basic concept that should be introduced is **"rule"**. In diKTat, a **"rule"** is the logic described in a class, which checks a certain paragraph of code style for compliance with the code. You should also know that **set** - is a well-defined collection of distinct objects, considered as an object in its own right. **Ruleset**, in turn, is a set of such "rules". **Abstract syntax tree(AST)** is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node[1] of the tree denotes a construct occurring in the source code. **CICD** - continuous integration (CI) and continuous delivery (CD) is a methodology that allows application development teams to make changes to code more frequently and reliably. **KDoc** - is the language used to document Kotlin code (the equivalent of Java's JavaDoc).

3 Kotlin

Kotlin is a cross-platform, statically typed, general-purpose programming language with type inference. Kotlin is designed to interoperate fully with Java, and the JVM version of Kotlin's standard library depends on the Java Class Library, but type inference allows its syntax to be more concise. Kotlin mainly targets the JVM, but also compiles to JavaScript (e.g. for frontend

[1] `https://www.ibm.com/support/knowledgecenter/SSZHNR_2.0.0/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTNode.html`

web applications using React) or native code (via LLVM), e.g. for native iOS apps sharing business logic with Android apps.

Kotlin has quickly skyrocketed in popularity. It's used by companies like Google, Square, Pinterest, Pivotal, Netflix and Atlassian. It's the fastest-growing programming language, according to GitHub, growing over 2,5 times in the past year. It was voted one of the five most loved languages, according to Stack Overflow. There are even meetups focused on Kotlin.

Kotlin is used in a lot of ways. For example it can be used for backend development using ktor framework (developed by JetBrains), and spring framework also has first-party support for kotlin (Spring is one of the most popular framework on Java for Web development). Kotlin/JS provides the ability to transpile your Kotlin code to JavaScript, as well as providing JS variant of kotlin standard library and interopability with existing JS dependencies, both for Node.js and browser. There are numerous ways that Kotlin/JS can be used. For instance, you can write frontend web applications using Kotlin/JS, write server-side and serverless applications using Kotlin/JS, create libraries for use with JavaScript and TypeScript. Support for multiplatform programming is one of Kotlin's key benefits. It reduces time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming.

Asynchronous or non-blocking programming is the new reality. Whether we're creating server-side, desktop or mobile applications, it's important that we provide an experience that is not only fluid from the user's perspective, but scalable when needed. There are many approaches to this problem, and Kotlin takes a very flexible one by providing Coroutine support as a first-party library kotlinx.coroutines with a kotlin compiler plugin and delegating most of the functionality to libraries, much in line with Kotlin's philosophy. As a bonus, coroutines not only open the doors to asynchronous programming, but also provide a wealth of other possibilities such as concurrency, actors, etc.

A coroutine is a concurrency design pattern that you can use on Android to simplify code that executes asynchronously. Coroutines(in a form of kotlinx.coroutines library and kotlin compiler plugin) were added to Kotlin in version 1.3 and are based on established concepts from other languages.

On Android, coroutines help to manage long-running tasks that might otherwise block the main thread and cause your app to become unresponsive. Over 50% of professional developers who use coroutines have reported seeing increased productivity. Coroutines enable you to write cleaner and more concise app code.

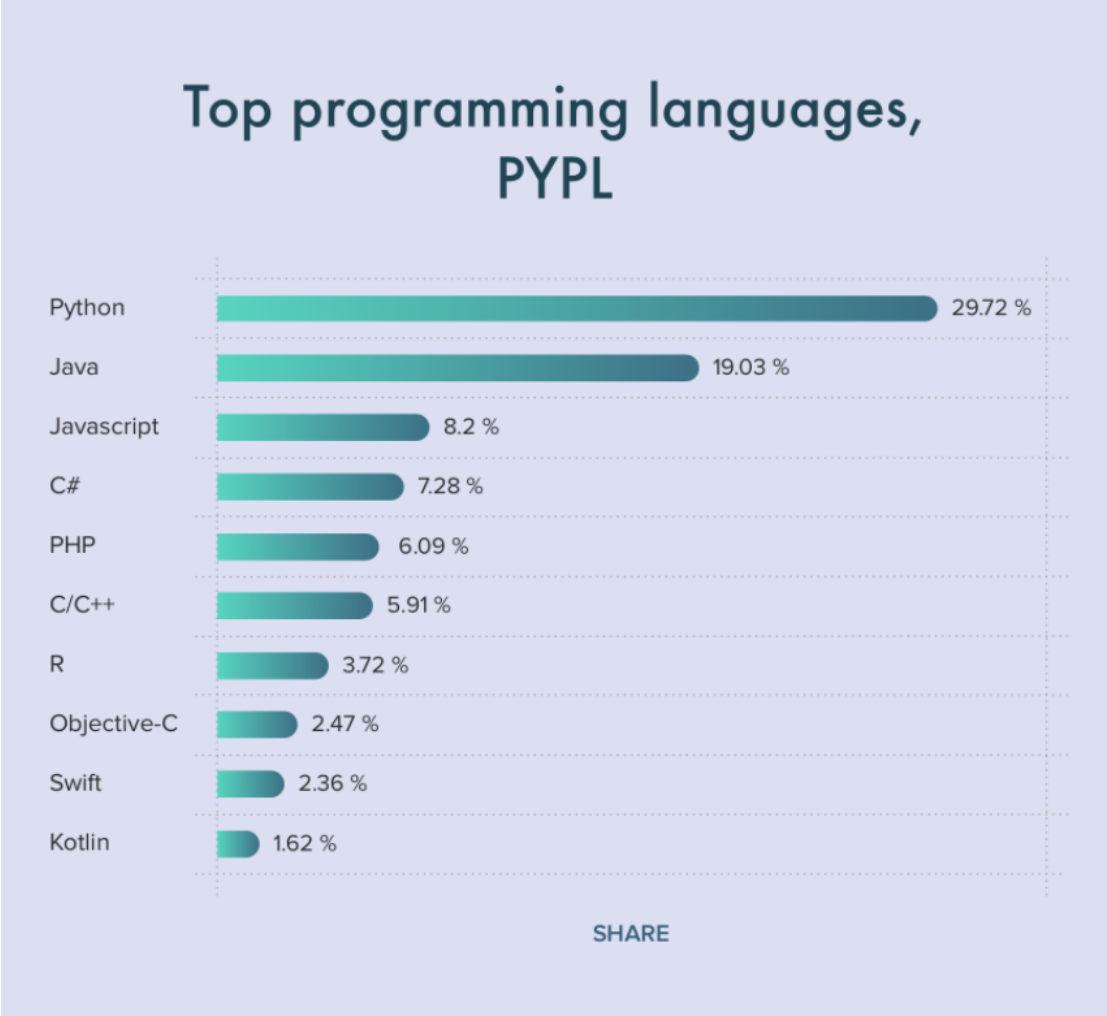
The state of Kotlin in 2020 (according to the latest Kotlin Census and statistical data)

- 4,7 million users
- 65% of users use Kotlin in production
- For 56% of users, Kotlin is their primary language, which means the main or only one they use at work
- 100+ people are on the Kotlin development team at JetBrains
- 350+ independent contributors develop the language and its ecosystem outside of JetBrains

Kotlin is widely used among Android developers, including open source OS, like HarmonyOS, that are compatible with Android. In 2019 Google announced that the Kotlin programming

language is now its preferred language for Android app developers. In the same year Stack Overflow stated that Kotlin is fourth most loved language in community. Nowadays there are over 60% of android developers who use Kotlin as their main language.

Kotlin's popularity can be explained by the rising number of Android users (last year, 124.4m in the USA) and, thus, Android-based devices. 80% of Kotlin programmers use the language to build Android apps, 31% for back-end applications, 30% for SDK/libraries. Kotlin is also inter-operable with Java, which allows developers to use all existing Android libraries in a Kotlin app. Kotlin is in the top of PYPL rating.



Overall, Kotlin is a modern language that gain it's popularity incredibly fast. It is mostly used by Android developers, but other "branches of programming" are gaining popularity as well, for example spring framework(the most popular Java framework) is supporting Kotlin. It supports both OO (object-oriented) and FP (function-oriented) programming paradigms. Since release 1.4 Kotlin claims to bring major updated every 6 month.

4 diKTat

4.1 What is diKTat?

DiKTat - is a formal strict code style (`https://github.com/cqfn/diKTat`) and a linter with a set of rules that implement this code style for Kotlin language. Basically, it is a collection of Kotlin code style rules implemented as AST visitors on top of KTlint framework (`https://github.com/pinterest/ktlint`). Diktat warns and fixes code style errors and code smells based on configuration file. DiKTat is a highly configurable framework, that can be extended further by adding custom rules. It can be run as command line application or with maven or gradle plugins. In this paper, we will explain how DiKTat works, describes advantages and disadvantages and how it differs from other static analyzers.

4.2 Why diKTat?

So why did we decide to create diKTat? We looked at similar projects and realized that they have defects and their functionality does not give you a chance to implement modern configurable code style. That's why we came to a conclusion that we need to create convenient and easy-to-use tool for developers. Why is it easy-to-use? First of all, diKTat has its own highly configurable ruleset. You just need to fill your own options on rules in ruleset or either use default one. Basically, ruleset is an yml file with a description of each rule. Secondly, there are a lot of developers that use different tools for building projects. Most popular are Maven and Gradle. DiKTat supports these ones and it also has cli. Finally, each developer has their own codestyle and sometimes they don't want static analyzers to trigger on some lines of code. In diKTat you can easily disable a rule.

5 Comparative analysis

5.1 About ktlint

Ktlint is a popular an anti-bikeshedding Kotlin linter with built-in formatter created by pinterest. It tries to capture (reflect) official code style from kotlinlang.org and Android Kotlin Style Guide and then automatically apply these rules to your codebase. Ktlint checks and can automatically fix code and it claims to be simple and easy to use. Ktlint has been developing since 2016 and from then on it has 3.8k stars, 299 forks and 390 closed PRs (at least on the moment of writing this whitepaper). There have been written over 15k lines of code. Ktlint has it's own ruleset, which divides on standard and experimental rules. Ktlint can be used as a plugin via Maven or Gradle. To configure rules in Ktlint you should modify .editorconfig file. But you can't configure specific rules, instead you can provide some common settings. In other words, ktlint has a "fixed hardcoded" codestyle that is not very configurable. Properties should be specified under **[*.kt,kts]**. If you want to implement your own rules you need to create a ruleset. Ktlint is using java's ServiceLoader to discover all available "RuleSets". ServiceLoader is used to inject your own implementation of rules for the static analysis. In this case ktlint becomes a third-party

dependency and a framework. Basically you should provide implementation of "RuleSetProvider" interface. Ktlint refers to article on **medium** [2]on how to create a ruleset and a rule.

A lot of projects uses ktlint as their code formatting tool. For example, OmiseGo [3] (currently rebranding to OMG Network) - a quite popular cryptocurrency.

5.2 About detekt

Detekt is a static code analysis tool. It operates on an abstract syntax tree provided by Kotlin compiler and, on top of that, they do complex analysis of code. However, this project is more focused on checking rather than fixing. Similarly to ktlint, it has it's own rules. Detekt uses wrapped ktlint to redefine rules as it's formatting rules. Detekt supports such features as code smell analysis, highly configurable rule sets, IntelliJ integration, third-party integrations for Maven, Bazel and Github actions, mechanism for suppression of their warnings with @Suppress annotation and many more. It is being developed since 2016 and today it has 3.2k stars, 411 forks and 1850 closed PRs. It has circa 45k lines of code.

Detekt is used in such projects as fountain or Kaspresso. "Fountain is an Android Kotlin library conceived to make your life easier when dealing with paged endpoint services" [4] and Kaspresso is a framework for UI testing on Android made by KasperskyLab [5].

5.3 About ktfmt

Ktfmt is a program that formats Kotlin code, based on google-java-format. It's development has started in Facebook in the end of 2019. It can be added to your project through a Maven dependency, Gradle dependency, IntelliJ plugin or you can run it through a command line. Ktfmt is not a configurable application, so to change any rule logic you need to download the project and redefine some constants. Ktfmt has 214 stars, 16 forks, 20 closed PRs and around 7500 lines of code.

5.4 About diKTat

Diktat as well as ktlint and detekt is a static code analysis tool. But diktat is not only a tool, but also coding convention (link) that in details describes all the rules that you should follow when writing a code on Kotlin. It's development has started in 2020 and at the time of writing this whitepaper diKTat has 130 stars and 12 forks. DiKTat operates on AST provided by kotlin compiler.

So why is diKTat better? First of all, we support much more rules than ktlint. Our ruleset includes more than 100 rules, that can both check and fix your code. Second, diKTat is configurable. A lot of rules have their own settings, and all of them can be easily understood. For example, you can choose whether you need a copyright, choose a length of line or you can configure your indentations. A little more about indentations, in this rule you can specify indentation

[2] https://medium.com/@vanniktech/writing-your-first-ktlint-rule-5a1707f4ca5b
[3] https://github.com/omgnetwork/android-sdk
[4] https://github.com/xmartlabs/fountain
[5] https://github.com/KasperskyLab/Kaspresso

size, whether you need a new line at the end or specify if you need aligned parameters. Third, diKTat is very easy to configure. You don't need to spend hours only to understand what each rule is doing. Our ruleset is a yml file, where each rule is commented out with the description. Last but not the least, diKTat can be used as a CI/CD tool in order to avoid merging errors in the code.

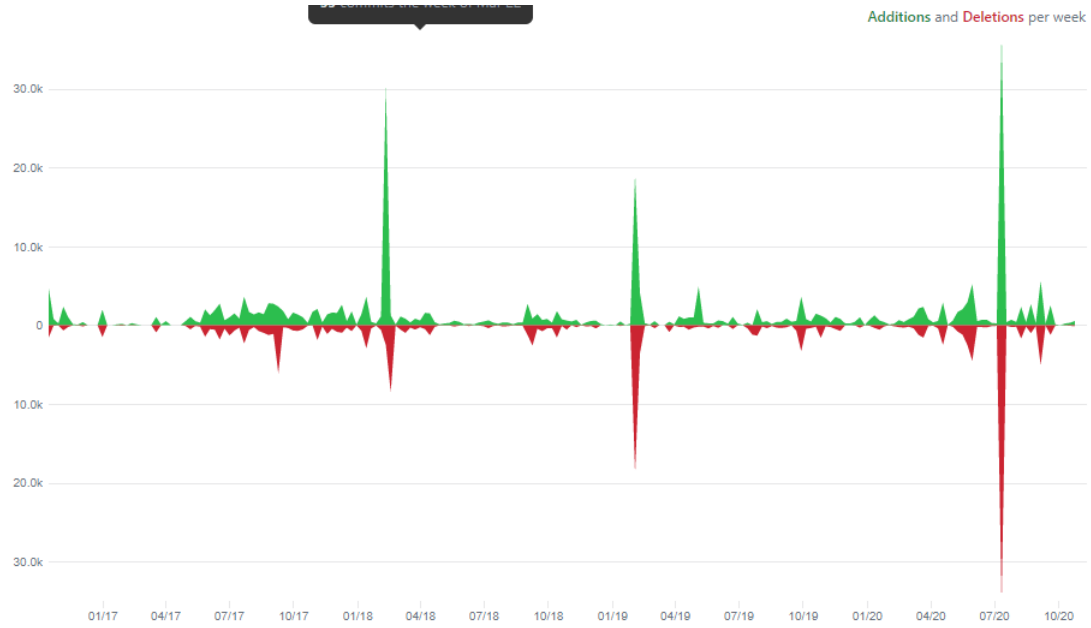
Overall it can find code smells and code style issues. Also it can find pretty unobvious bugs by complex AST analysis.

5.5 A few words about Jetbrains

Jetbrains created one of the best IDEs for Java and Kotlin called IntelliJ. This IDE supports a built-in linter. However it is not a well-configurable tool, you are not able to specify your own coding convention and it is not useful for CI/CD as it is highly coupled with UI. Unfortunately such static analysis is not so effective as it cannot prevent merging of the code with bugs into the repository. As experience shows - many developers simply ignore those static analysis errors until they are blocked from merging their pull requests.

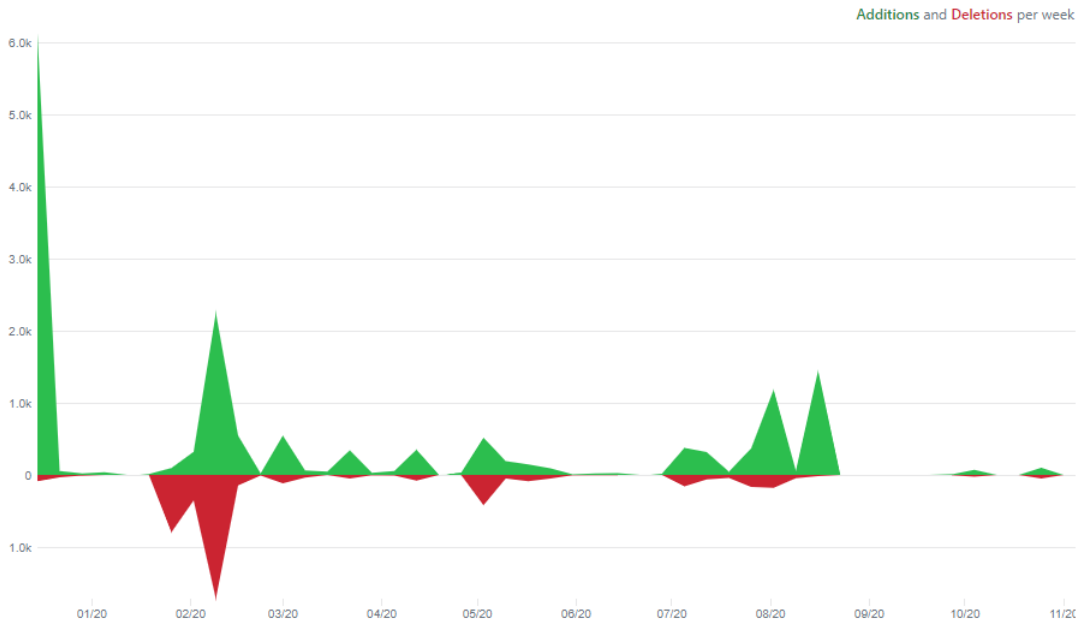
5.6 Graphics

5.6.1 Detekt Code Frequency



5.6.2 Ktlint Code Frequency

5.7 Ktfmt Code Frequency



5.7.1 DiKTat Code Frequency

5.8 Summary

Comparing table				
	diKTat	ktlint	detekt	ktfmt
starting year	2020	2016	2016	2019
stars	130	3.2k	3.8k	214
forks	12	299	411	16
closed PRs	226	390	1850	20
lines of code	22k	15k	45k	7,5k
number of rules	>100	≈ 20	>100	≈ 10

6 How does it work

Diktat does AST-analysis. This means that for internal representation (IR) it uses Abstract Syntax Tree that was created from the parsed code by the kotlin-compiler. This chapter describes how diktat works

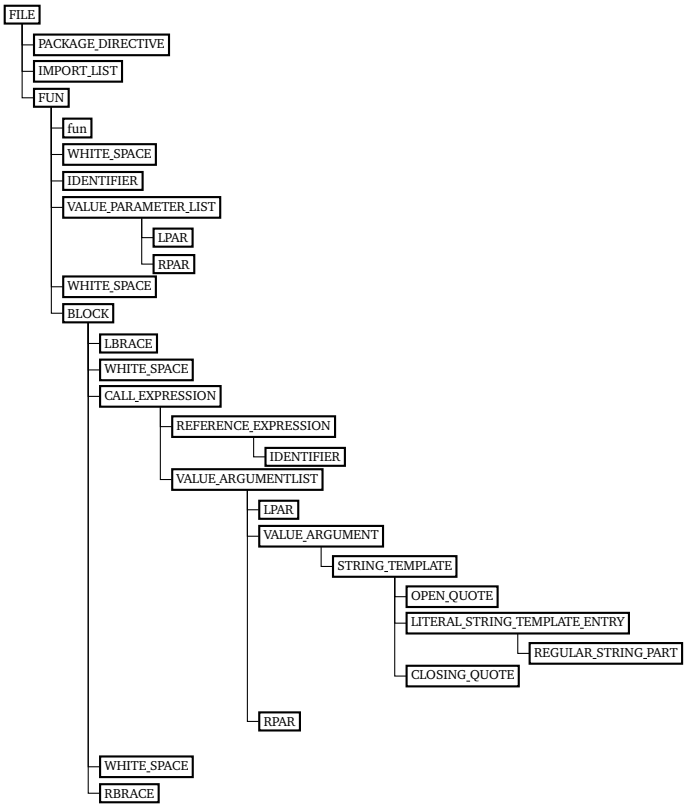
6.1 ktlint

To quickly and efficiently analyze the program code, you first need to transform it into a convenient data structure. This is exactly what ktlint does - it parses plain text code into an abstract syntax tree. In ktlint, this happens in the *prepareCodeForLinting*[6] method. This method uses kotlin-compiler-embeddable library to create a root node of type FILE. For example, this simple code:

```
1  fun main() {
2    println("Hello World")
3  }
```

Listing 1: Simple function.

will be converted into this AST:



If there are error elements inside the constructed tree, then the corresponding error is displayed. If the code is valid and parsed without errors, for each rule in the ruleset, the *visit* method is called to which the root node itself and its "children" are sequentially passed. When you run program, you can pass flags to ktlint - one of them is "-F". This flag means that the rule will not only report an error, but try to fix it.

[6] https://github.com/pinterest/ktlint/blob/master/ktlint-core/src/main/kotlin/com/pinterest/ktlint/core/KtLint.kt

6.2 diKTat

Another feature of ktlint is that at startup you can provide a JAR file with additional ruleset(s), which will be discovered by the ServiceLoader and then all nodes will be passed to these rules. This is diKTat! DiKTat is a set of easily configurable rules for static code analysis. The set of all rules is described in the *DiktatRuleSetProvider*[7] class. This class overrides the *get()* method of the *RuleSetProvider*[8] interface, which returns a set of rules to be "traversed". But before returning this set, the configuration file, in which the user has independently configured all the rules, is read. If there is no configuration file, then a warning will be displayed and the rules will be triggered in according with the default configuration file. Each rule must implement the *visit* method of the abstract Rule class, which describes the logic of the rule.

//TODO: add comments

```
class SingleLineStatementsRule(private val configRules: List<RulesConfig>) : Rule(
    "statement") {

    companion object {
        private val semicolonToken = TokenSet.create(SEMICOLON)
    }

    private lateinit var emitWarn: ((offset: Int, errorMessage: String,
    canBeAutoCorrected: Boolean) -> Unit)
    private var isFixMode: Boolean = false

    override fun visit(node: ASTNode,
                       autoCorrect: Boolean,
                       emit: (offset: Int, errorMessage: String,
    canBeAutoCorrected: Boolean) -> Unit) {
        emitWarn = emit
        isFixMode = autoCorrect

        checkSemicolon(node)
    }

    private fun checkSemicolon(node: ASTNode) {
        node.getChildren(semicolonToken).forEach {
            if (!it.isFollowedByNewline()) {
                MORE_THAN_ONE_STATEMENT_PER_LINE.warnAndFix(configRules, emitWarn,
     isFixMode, it.extractLineOfText(),
                        it.startOffset, it) {
                    if (it.treeParent.elementType == ENUM_ENTRY) {
                        node.treeParent.addChild(PsiWhiteSpaceImpl("\n"), node.
    treeNext)
                    } else {
                        if (!it.isBeginByNewline()) {
                            val nextNode = it.parent({ parent -> parent.treeNext
    != null }, strict = false)?.treeNext
                            node.appendNewlineMergingWhiteSpace(nextNode, it)
```

[7] https://github.com/cqfn/diKTat/blob/v0.1.3/diktat-rules/src/main/kotlin/org/cqfn/diktat/ruleset/rules/DiktatRuleSetProvider.kt

[8] https://github.com/pinterest/ktlint/blob/master/ktlint-core/src/main/kotlin/com/pinterest/ktlint/core/RuleSetProvider.kt

```
                            }
                            node.removeChild(it)
                        }
                    }
                }
            }
        }
}
```

Listing 2: Example of rule.

The example above describes the rule in which you cannot write more than two statements on one line. The list of configurations is passed to the parameter of this rule so that the error is displayed only when the rule is enabled (further it will be described how to enable and disable the rule). The class fields and the *visit* method are described below. The first parameter in method is ASTNode - the node that we got in the parsing in ktlint. Then a check occurs: if the code contains a line in which more than one statement per line and this rule is enabled, then the rule will be executed and, depending on the mode in which the user started ktlint, the rule will either simply report an error or fix it. In our case, when an error is found, the method is called to report and fix the error - *warnAndFix()*. Warnings that contain similar logic (e.g. regarding formatting of function KDocs) are checked in the same Rule. This way we can parse similar parts of AST only once. Whether the warning is enabled or disabled is checked at the very last moment, inside *warn()* or *warnAndFix()* methods. Same is true for suppressions: right before emitting the warning we check whether any of current node's parents has a Suppress annotation.

6.3 Examples of unique inspections

As already described above, diKTat has more rules than existing analogues, therefore, it will find and fix more errors and shortcomings and, thereby, make the code cleaner and better. To better understand how much more detailed the diKTat finds errors, consider a few examples:

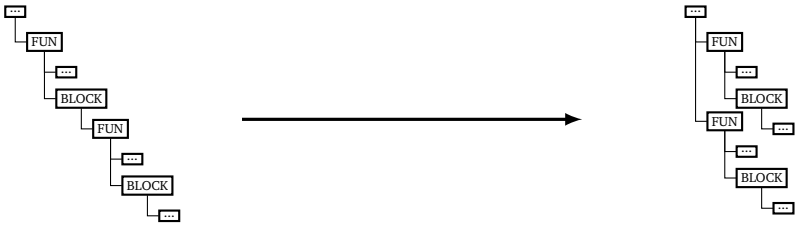
1. **Package** In diKTat, there are about 6 rules only for package (examples). For comparison: detekt has only one rule, where the package name is simply checked by a pattern, in ktlint there is no.
2. **KDoc** KDoc is an important part of good code to make it easier to understand and navigate the program. In diKTat there are 15 rules on KDoc, in detekt there are only 7. Therefore, diKTat will make and correct KDoc in more detail and correctly. Examples of rules that have no analogues: examples
3. **Header** Like KDoc, header is an essential part of quality code. DiKTat has as many as 6 rules for this, while detekt and ktlint do not. (Examples)

There are also many unique rules that no analogues have, here are some of them:

1. **COMMENTED_OUT_CODE** – This rule performs checks if there is any commented code.
2. **FILE_CONTAINS_ONLY_COMMENTS** – This rule checks file contains not only comments.

3. **LOCAL_VARIABLE_EARLY_DECLARATION** – This rule checks that local variables are declared close to the point where they are first used.

4. **AVOID_NESTED_FUNCTIONS** - This rule checks for nested functions and warns and fixes if it finds any. An example of changing the tree when this rule is triggered and diKTat is run with fix mode:.



5. **FLOAT_IN_ACCURATE_CALCULATIONS** - Rule that checks that floating-point numbers are not used for accurate calculations.

7 Killer-Feature

As described above, diKTat is very configurable and user-friendly. But these are not all of its advantages and features. Below will be presented and described unusual and important killer-features of diKTat.

7.1 Configuration file

It's worth starting with the configuration file. This is a file in which the user can manually turn rules on and off or configure the rules settings. Below is one of the rules in the configuration file.

```
name: FILE_IS_TOO_LONG
        enabled: true
        configuration:
                maxSize: '2000'
                ignoreFolders: ' '
```

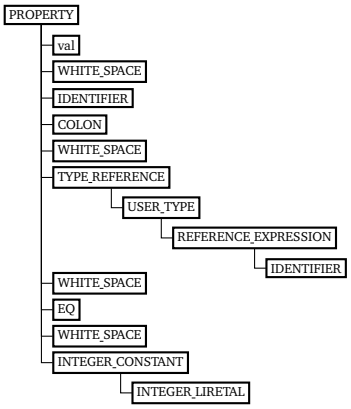
Each rule in this file has 3 fields: name - the name of the rule, enabled - whether the rule is enabled or disabled (all rules are enabled by default), configuration - parameters for the rule. With the first two, everything is obvious. The third parameter is less obvious. The configuration is a set of "properties" to configure this rule. For example, for a rule "FILE_IS_TOO_LONG", that checks the number of lines in a Kotlin file, the user can configure the maximum number of lines allowed in the file - by changing the "maxSize" in the configuration, or the user can specify paths to folders that do not need to be checked - by writing the path in "ignoreFolders".

7.2 Create ASTNode

Another feature is a method that allows you to construct an abstract syntax node from text. This algorithm can parse the code even partially, when you do not need to save the hierarchy of the file (with imports/packages/classes). For example it can parse and provide you a sub-tree for these lines of code:

```
1  val nodeFromText: ASTNode = KotlinParser().createNode("val age: Int = 21")
```

Listing 3: Example of creating node.



As you can see in the examples, we pass to the method the text of the source code that we want to transform and the flag, which is set to false by default. The flag should be set to true in order to immediately build a tree with a root node of the FILE type. What's going on inside this method? First of all, the system properties are set (for example: set "idea.io.use.nio2" to true). Further in the method, the text is checked. If the text of the code contains such keywords as import or package, then the method builds a tree with a root node of the FILE type, otherwise it tries with a different root type. In both cases, at the end, if the tree contains an ERROR_ELEMENT type of node, it means that an error was made in the code and the method was unable to build the tree and, therefore, throws an exception.
This helps us to implement such complex inspections like the detection of commented code, helps easily fix the code without manually building sub-trees in visitors

7.3 "SUPPRESS annotation"

What if the user wants one of the diKTat rules not to check a piece of code? The *SUPPRESS* annotation will help with this. This annotation can be supplied to ignore a certain rule. For instance, if run this code:

```
1  /**
2 * This is example
3 */
4
5 package org.cqfn.diktat
6
```

```
/**
* Simple class
*/
class User(private val name: String, private val age: Int) {
  /**
  * Function with incorrect name
  *
  * @return is username longer than age
  */
  fun IsInCoRrEcTnAMe() = name.length > age
}
```

Listing 4: Function with incorrect name.

There will be warning:

```
[FUNCTION_NAME_INCORRECT_CASE] function/method name should be in lowerCamelCase
```

But if there is a *@SUPPRESS* before this method, then there will be no errors at run.

```
  /**
* This is example
*/

package org.cqfn.diktat

/**
* Simple class
*/
@Suppress("FUNCTION_NAME_INCORRECT_CASE")
class User(private val name: String, private val age: Int) {
  /**
  * Function with incorrect name
  *
  * @return is username longer than age
  */
  fun IsInCoRrEcTnAMe() = name.length > age
}
```

Listing 5: Function with incorrect name, but with suppress.

The example shows that the method has SUPPRESS annotation. Therefore, the FUNCTION_NAME_INCORRECT_CASE rule will be ignored on this method and there will be no error. The search method for a given annotation goes up recursively to the root element of type FILE, looking for the annotation. This means that Suppress can be placed not only in front of knowingly incorrect code, but also at the upper levels of the abstract tree. In our example, the annotation is not in front of the method, but in front of the class and still works. Also, you can put several annotations:

```
@set:[Suppress("WRONG_DECLARATION_ORDER") Suppress("IDENTIFIER_LENGTH")]
```

Listing 6: Function with incorrect name, but with suppress.

7.4 WEB

Also worth mentioning is the existence of a web version of diKTat.. This is a very handy tool that can be used quickly, and most importantly, it is very simple. The link can be found in or you can find it in ”How to use diKTat” chapter or in ktlint project as reference.[9]

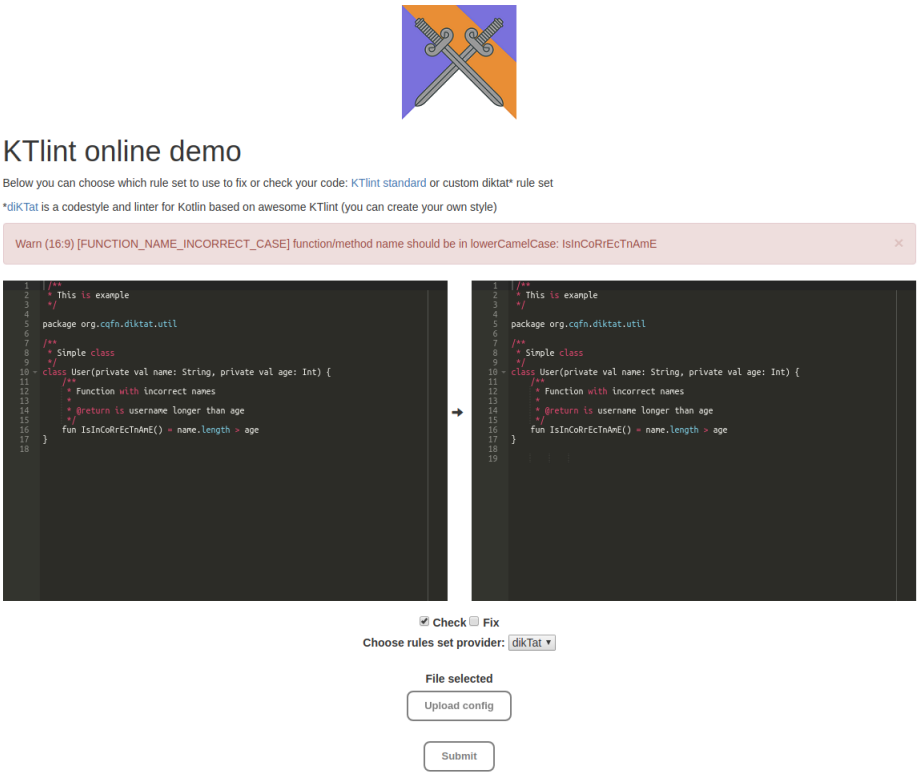


Figure 1: Example of web application

7.5 Validation rules

As it has been mention earlier, diktat has a highly customizable configuration file, but manually editing it is error-prone, for example, name of the rule can be incorrect due to a typo. Diktat will validate configuration file on startup and suggest the closest name based on *Levenshtein* method.

7.6 CI-CD

One of the most important parts of open source project is CI/CD pipeline as it brings considerable benefits to the entire software development process. It improves code quality, customer satisfaction and reduces costs, including time, of making new features.

[9] `https://github.com/pinterest/ktlint#online-demo`

8 How to use diKTat

8.1 CLI-application

You can run diKTat as a CLI-application by installing it first. You can find detailed instructions on how to install and run on different OS on github (`https://github.com/cqfn/diKTat/blob/master/README.md#run-as-cli-application`). After the run, errors will be found and displayed. Each error consists of a rule name, a description of the rule so that the user can understands the error, the line and column number where the error was found.

8.2 Plugins

Alternatively, you can add a diktat maven or gradle plugin directly to the project: detailed instructions for maven can be found here - `https://github.com/cqfn/diKTat/blob/master/README.md#run-with-maven`and for gralde - `https://github.com/cqfn/diKTat/blob/master/README.md#run-with-gradle-plugin`.

8.3 Configuration file

As described above, diKTat has a configuration file. Note that you should place the *diktat-analysis.yml* file containing the diktat configuration in the parent directory of your project when running as a CLI application. Diktat-maven-plugin and diktat-gradle-plugin have a separate option for configuration file path.

8.4 WEB

Of course, the easiest way to use it without any downloads or installations is the web version of the app. You can try it by following the link `https://ktlint-demo.herokuapp.com/demo`. Web app supports both checking and fixing, using either ktlint or diktat ruleset. For diktat you can also upload a custom configuration file.

9 Examples

Describe how does diKtat work on real projects

10 Conclusion & Future Work

Diktat is a static code analyzer that finds and fixes code style inconsistencies. DikTat is configurable and ..., which distinguishes it from analogues. We offer many convenient ways to use diktat in projects. ...

References

[1] T. Kremenek, “Finding software bugs with the clang static analyzer,” 2008.

11 Appendix

4-6 sentences.