Automating Tool Configurations for Coding Standards via DSL-Driven, LLM-Based Transpilation

Anonymous Author(s)

Appendices

A CODING STANDARDS AND TOOLS IN PRACTICE

A.0.1 Motivation. We are first interested in understanding the trends of coding standards and their corresponding tools in real-world software development.

A.0.2 Approach. We investigate trends in coding standards and tools in software development on GitHub. Building on prior research and developer practices [2, 6, 7, 9, 12–14, 17], we use keyword-based search method to collect projects related to coding standards and their corresponding tools. To ensure for project quality, we consider only projects with at least five stars. For coding standard-related projects, we search using keywords such as coding standard, code standard, convention, and style guide, then manually verify their relevance, identifying 1,066 projects. For tool-related projects, we use keywords like coding standard tool, code standard tool, convention tool, style guide tool, and linter, followed by manual verification, yielding 1,692 projects.

To examine the long-term popularity of coding standards and tools, we group the projects by their creation year. To determine whether they are actively maintained, we check if projects have commits within the current year. Commits serve as an important indicator of project maintenance, reflecting their continued development and engagement.

A.0.3 Result. Figure 1 shows the number of coding standards and tool projects from 2009 to 2024. By 2024, there are 1,066 coding standard projects, with new projects created annually ranging from 2 ~154. The number of new projects gradually increased from 2009 but slowed after 2016. Despite this, 20% (218 out of 1,066) of projects are still actively updated in 2024. Typically, coding standards remain stable as they are tied to specific organizational or project requirements, but the fact that 20% are still maintained suggests they continue to evolve in response to changes in development practices and technology.

For tool projects related to coding standards, by 2024, there are 1,692 total projects, with new projects ranging from 4 ~202 annually. Similar to coding standards, new tool projects gradually grew until 2015, fluctuated between 2016 and 2020, and began to slow down after 2021. Despite this, 47% of tool projects (787 out of 1,692) remain actively maintained in 2024. Unlike coding standards, tools for coding standards are more dynamic, evolving rapidly with new features, technologies, and practices.

Given the large number of coding standards and tools, along with their frequent updates, developers must continuously track changes, making configuration generation an ongoing effort rather than a one-time task.

Summary: Projects related to coding standards and their corresponding tools on GitHub are continuously increasing and being updated.

B DSL FOR CODING STANDARDS AND TOOL CONFIGURATIONS

B.0.1 Motivating Examples for DSL Design. Coding standards and tool configurations both represent sets of coding rules, though they are expressed in different forms. Coding standards are typically written in natural language (NL), while tool configurations often use machine-readable formats like XML.

Coding standards are typically expressed in natural language, which is tool-agnostic and readable but lacks structure and precision. A NL sentence of coding standards may or may not contain rules and can embed multiple layers of information, such as rule types, objects being checked, and constraint types. For instance, in Figure 2, the sentences in the *NL box* illustrate this issue. In the Google Java coding standard, sentence *i* contains no rules, while sentence *j* conveys two rules: “should always” and “must” indicate mandatory rules; “block tag” and “these tags” refer to the four types of block tags being checked; “appear in this order” and “not have” imply order and negation constraint relationships.

In contrast, tool configurations are typically expressed in machine-readable formats, which are structured and precise but tool-dependent and often too vague to convey clear meanings. In the *Machine-Readable Format box* of the Figure 2, the Checkstyle configuration for sentence *j* clearly defines the rules, with module names specifying behavior and property names controlling specifics. However, the module names, property names, and values are often ambiguous, making their exact meaning unclear. Moreover, the format and configuration settings are specific to Checkstyle, while other linters adopt different formats and configuration settings.

To provide a tool-agnostic, readable, structured, and precise representation, we propose a domain-specific language (DSL), as shown in the *blue box labeled domain-specific language.* The corresponding DSL representations of sentence *j* are “Mandatory: Order of [BlockTag] is [@param, @return, @throws, @deprecated];” and “Mandatory: No [EmptyDescription] for [@param, @return, @throws, @deprecated]”. Such rule representations clearly express each rule: the prefix “Mandatory:” denotes that the rule must be satisfied, the objects being checked are enclosed in “[ ]” to differentiate them from the constraint relationship, and “Order of” and “No” represent different constraint relationships—order and negation.

B.0.2 Information Source for Designing DSL. To design an expressive and universal DSL for coding standards and tool configurations. We first determine research objects of coding standards and tool configurations. Java, with its mature ecosystem and rigorous coding practices [5], represents a language with long-standing conventions, making it an ideal choice. Given Google’s coding standards’ high recognition across software projects and the industry [3], we

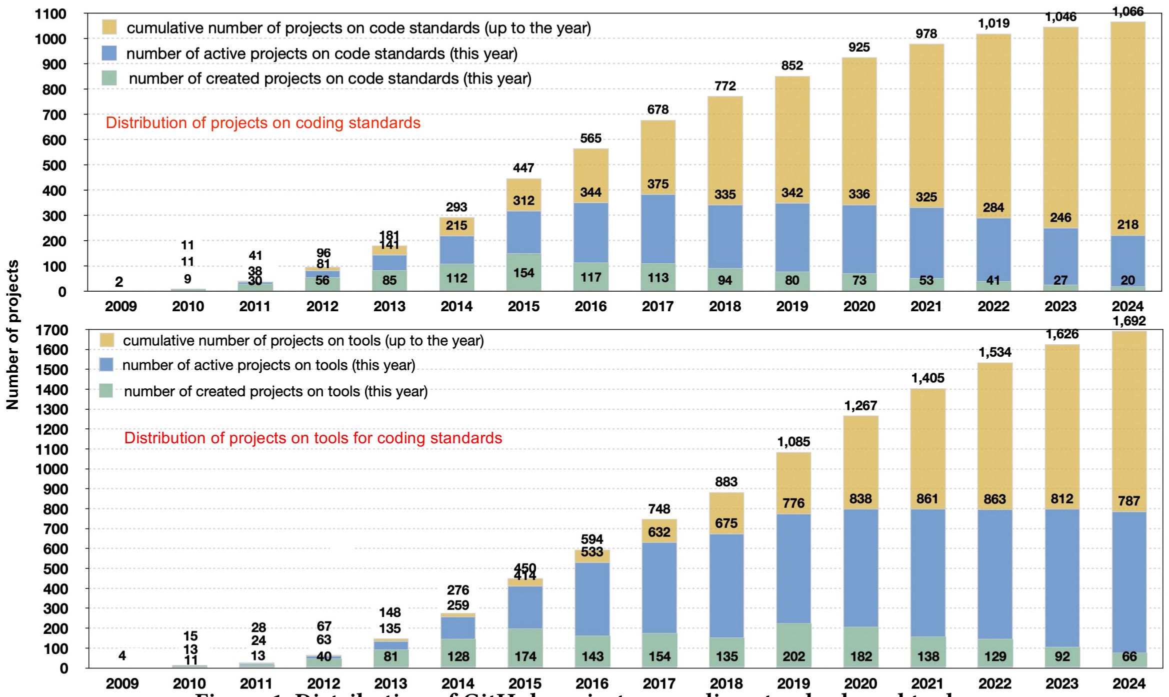

Figure 1: Distribution of GitHub projects on coding standards and tools

Natural Language (NL)

Google Java Style Guide:
- **Sentence i:** Sometimes there is more than one reasonable way to convert an English phrase into camel case.
- **Sentence j:** The standard block tags @param, @return, @throws, @deprecated should always appear in this order, and these tags must not have an empty description.

Explanation:
- **Sentence i:** it does not contain rules.
- **Sentence j:** it contains rules but **mixes different level information** such as the number of rules, rule types, objects being checked by rules, and corresponding constraint types of rules.
  - the sentence contains **two rules**;
  - "should always" and "must" indicate that the two rules are **mandatory rules**;
  - "block tag" and "these tags" refer to the **four types of block tags being checked**;
  - "appear in this order" and "not have" imply **constraint relationships**, such as order and negation.

NL Summary:
Good: general; readable **Bad:** too free → lacking structure and precision

Tool Configuration Format (e.g., XML)

Checkstyle Configuration:
```
<module name='AtclauseOrder'>
   <property name="tagOrder" value="@param, @return, @throws,
    @deprecated"/> </module>
<module name='NonEmptyAtclauseDescription'>
   <property name="tokens" value="PARAM_LITERAL,
     RETURN_LITERAL, THROWS_LITERAL,
     DEPRECATED_LITERAL"/> </module>
```

Explanation:
- The module names, property name and values are **too vague and abstract**, making it difficult to understand their specific meanings.
- The **format** and **configuration settings**, such as module names, are **specific to** Checkstyle, while other linters use distinct formats and settings.

Tool Configruation Format Summary:
Good: structured; precise **Bad:** tool-dependent; hard to read

Domain-Specific Language (DSL)

DSL Representation:
Mandatory: Order of [BlockTag] is [@param, @return, @throws, @deprecated] ;
Mandatory: No [EmptyDescription] for [@param,@return,@throws,@deprecated]

Explanation:
- **Each rule** is represented **independently**.
- The prefix "Mandatory:" denotes the **rule type**, indicating it must be satisfied.
- "Order " and "No" denote different **constraint relationships**: order and negation.
- **Checking objects** are enclosed in [], distinguishing them from the **constraint relationship**.

DSL Summary:
Good: general; readable; structured; precise

The DSL We Designed:
```
RuleSet      ::= Rule [';' Rule]*
Rule         ::= ['Optional:' | 'Mandatory:'] Constraint [ExceptRule]*
Constraint   ::= TermList [Operator TermList]*
               | 'No' Constraint
               | 'Order of' TermList ['is' | 'is not'] TermList
               | 'Number of' TermList Operator TermList
               | 'if' Constraint 'then' Constraint
ExceptRule   ::= 'Except' Constraint
Operator     ::= [.]+  (e.g., 'is' | 'is not' | 'have' | 'after' |  '=' | 'Add ' | '...' )
TermList     ::= Term [', ' Term]*
Modifier     ::= Word (e.g., 'some' | 'each' | 'all' | 'first' | 'last' | '...')
Term         ::= '[' PLterm ']'
               | Modifier* PLterm
               | PLterm 'of' PLterm
Word         ::= [a-zA-Z]+
PLterm       ::= [.]+
```
Note: PLterm *denotes terminology in programming languages like Java;*
'. ' *means to match any single character except for newline.*
'...' *indicates that more words can be included if needed;*
'*' *means zero or more repetitions;* '+' *means one or more repetitions.*

Figure 2: Comparison of different representations of coding rules

choose the Google Java style guide. To enforce these standards, we select Checkstyle [1] for Java coding standards. Checkstyle is highly configurable, widely used, and have been studied extensively [8, 10, 16, 17]. To understand tool configurations, we refer to Checkstyle documentation [4], which provides detailed information about tool configurations.

B.0.3 Process of Designing DSL. We use a card sorting approach [15] to define DSL by analyzing a sample of 320 sentences with a confidence level of 95% and a confidence interval of 5 drawn from all sentences of the Google JavaScript style guide and Checkstyle documentation. The process of designing DSL has two iterations and we used two evaluators. In the first iteration, we randomly sample 175 sentences with a confidence level of 95% and a confidence interval of 5 from 320 sentences. Two authors first independently represent each sentence using their designed DSLs, and then they discuss to construct a DSL. In the second iteration, two of the authors independently represent remaining sentences with the DSL. If the DSL is not enough to represent the rule of a sentence, they annotate the sentence with a brief description. They find that there are no sentences that cannot be represented using the DSL. We use Cohen's Kappa measure [11] to examine the agreement between the two authors. The Kappa value is 0.7, which indicates a high agreement between two authors. Finally, two authors discuss the disagreements to reach an agreement.

B.0.4 DSL for Coding Standards and Tool Configurations. The DSL for coding standards and tool configurations, designed by referencing Google's Java coding standards along with Checkstyle documentation, is shown in Figure 2.

• RuleSet represents sets of Rules, where individual rules are separated by semicolons.

• A Rule comprises a RuleType, a Constraint, and optionally any number of ExceptRule. The RuleType can be 'Mandatory' or 'Optional', indicating whether the rule must be followed or is optional. The ExceptRule specifies exceptions to a Constraint. Constraint is one of the following six types:

(1) TermList [Operator TermList]*: Denotes the relationship that TermList must satisfy, expressed via operators. For example, the Checkstyle documentation, "Allow loops with empty bodies" is represented in the DSL as "Optional: [body] of [LoopStatement] is [Empty]", where the operator is "is".

(2) 'No' Constraint: The Constraint is prohibited. For example, in the Figure 2, the NL description of the Google Java coding standard, "these tags must not have an empty description" is represented in the DSL as "Mandatory: No [EmptyDescription] for [@param, @return, @throws, @deprecated]".

(3) 'Order of' TermList ['is' | 'is not'] TermList: Defines the ordering requirements where one TermList should or should not follow another TermList. For example, in the Figure 2, the NL description of the Google Java coding standard, "the standard block tags @param, @return, @throws, @deprecated should always appear in this order" is represented in the DSL as "Mandatory: Order of [BlockTag] is [@param, @return, @throws, @deprecated]".

(4) 'Number of' TermList Operator TermList: Specifies constraints on the quantity of TermList. For example, the Google Java coding standard "each annotation is listed on a line of its own (that is, one annotation per line)" is represented in DSL as: "Mandatory: Number of [Annotation] = 1 for each [Line]".

(5) 'if' Constraint 'then' Constraint: Represents a constraint should be satisfied when the another constraint satisfied. For example, the Checkstyle documentation, "Allow to ignore enums when left curly brace policy is EOL" is represented in DSL as: "Optional: if [LeftCurlyPolicy] is [EOL] then [Enum] of [LeftCurly] is not [EOL]".

• Operator denotes the relationship between TermList and consists of any sequence of characters except for newline, with one or more characters allowed. The examples provided, such as 'is', 'is not', 'have' and others, are specific instances of operators that fall into this category.

• TermList is a list of Terms separated by commas.

• A Modifier is a single word (e.g., 'some,' 'each,' 'all,' 'first,' 'last,' etc.) that provides additional context or restrictions for a PLterm.

• Term can be one PLterm representing terminology used in programming languages like Java. It can also be a composite structure formed by PLterm connected with Modifier or 'of'.

• Word consists of alphabetic characters.

• PLterm represents terminology specific to programming languages, such as "BlockTag" and "@param" in the DSL representation of the Figure 2. It can be any sequence of characters, excluding newlines.

REFERENCES

[1] 2024. *CheckStyle.* https://github.com/checkstyle/checkstyle
[2] 2024. *ESLint.* https://github.com/eslint/eslint
[3] 2024. *Google Style Guide.* https://github.com/google/styleguide
[4] 2024. *Offical Documentation of Checkstyle.* https://checkstyle.sourceforge.io/checks.html
[5] Mohammad MA Abdallah and Mustafa M Al-Rifaee. 2017. Java standards: A comparative study. *International Journal of Computer Science and Software Engineering* 6, 6 (2017), 146.
[6] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering.* 281–293.
[7] Cathal Boogerd and Leon Moonen. 2008. Assessing the value of coding standards: An empirical study. In *2008 IEEE International conference on software maintenance.* IEEE, 277–286.
[8] Stephen H Edwards, Nischel Kandru, and Mukund BM Rajagopal. 2017. Investigating static analysis errors in student Java programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research.* 65–73.
[9] Boryana Goncharenko and Vadim Zaytsev. 2016. Language design and implementation for the domain of coding conventions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering.* 90–104.
[10] Benjamin Loriot, Fernanda Madeiral, and Martin Monperrus. 2022. Styler: learning formatting conventions to repair Checkstyle violations. *Empirical Software Engineering* 27, 6 (2022).
[11] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica* (2012).
[12] Naoto Ogura, Shinsuke Matsumoto, Hideaki Hata, and Shinji Kusumoto. 2018. Bring your own coding style. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 527–531.
[13] Andrew J Simmons, Scott Barnett, Jessica Rivera-Villicana, Akshat Bajaj, and Rajesh Vasa. 2020. A large-scale comparative analysis of coding standard conformance in open-source data science projects. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* 1–11.
[14] Michael Smit, Barry Gergel, H James Hoover, and Eleni Stroulia. 2011. Code convention adherence in evolving software. In *2011 27th IEEE International Conference on Software Maintenance (ICSM).* IEEE, 504–507.
[15] Donna Spencer. 2009. *Card sorting: Designing usable categories.* Rosenfeld Media.
[16] Eric Torunski, M Omair Shafiq, and Anthony Whitehead. 2017. Code style analytics for the automatic setting of formatting rules in ides: A solution to the tabs vs. spaces debate. In *2017 Twelfth International Conference on Digital Information Management (ICDIM).* IEEE, 6–14.
[17] Chau Chin Yiu. 2023. Checkstyle for Legacy Applications [J]. *Itestra De* (2023).